UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL

Mejora del rendimiento de la reflexión estructural mediante técnicas de compilación JIT

Presentada por

José Manuel Redondo López

Para obtención del título de Doctor por la Universidad de Oviedo

Dirigida por el

Profesor Doctor D. Francisco Ortín Soler

Oviedo, 27 de Enero de 2007

AGRADECIMIENTOS

Al director de esta tesis, Francisco Ortín Soler, por sus consejos, por su incuestionable ayuda para centrar y llevar adelante este trabajo de la mejor forma y porque sin su intervención nada de esto hubiese sido posible.

A toda mi familia y a Sabrina, por estar apoyándome siempre que lo necesitaba y no dejarme tirar la toalla en los peores momentos.

A Luís y Benjamín, por su aportación para llevar este barco a buen puerto.

A mis compañeros de trabajo, por su interés y sus palabras de ánimo.

A todos aquéllos que no se esconden cuando se trata de echar una mano.

A todos los que se levantan cada día con la idea de ser mejores personas.

A todos los que se niegan a ser vencidos por los reveses de la vida.

Por todo aquél que me ha enseñado algo en mi vida, y por todo aquél que me lo enseñará en un futuro.

Y por supuesto, por cualquiera que crea que debería figurar en esta sección y no aparece (no se puede estar en todo…)

Gracias a todos

Qué equivocado estaba en muchas cosas y cuánto he aprendido con esta labor. Sólo espero que en el futuro pueda seguir aprendiendo y equivocándome, pues sin duda eso me hará una persona mejor.

RESUMEN

Los lenguajes dinámicos son aquéllos que permiten analizar y modificar la estructura, comportamiento y entorno de los programas en tiempo de ejecución. La principal ventaja ofrecida por estos lenguajes es permitir construir *software* que pueda adaptarse a posibles cambios que ocurran dinámicamente en sus requisitos, soportando características como metaprogramación, reflexión, reconfiguración dinámica y distribución, entre otras.

Varios lenguajes dinámicos, como *Python*, *Ruby* o *Dylan*, están aumentado actualmente su popularidad como medio de desarrollo de aplicaciones que requieran un alto grado de flexibilidad. Estos lenguajes se ejecutan habitualmente sobre una máquina virtual implementada como un intérprete, aprovechando las ventajas que éstas ofrecen. Ello implica que el rendimiento de las aplicaciones realizadas con estos lenguajes es menor que el ofrecido por los lenguajes "estáticos". Adicionalmente, el mayor número de operaciones que los lenguajes dinámicos deben realizar durante la ejecución, para proporcionar su elevado grado de flexibilidad, supone un coste adicional en su rendimiento.

Existe un campo de investigación dedicado a mejorar el rendimiento de las aplicaciones que se ejecutan sobre máquinas virtuales. Técnicas como la compilación "Justo a Tiempo" (*Just in Time*, *JIT*) o la optimización adaptativa han supuesto una mejora de rendimiento considerable, permitiendo el uso de estas plataformas para el desarrollo de sistemas comerciales, como sucede en el caso de los lenguajes *Java* o *C#*. Por tanto, para mejorar el rendimiento de los lenguajes dinámicos, esta tesis emplea el mismo principio que ha sido válido para la implementación de máquinas virtuales "estáticas". Partiendo de una máquina virtual profesional con compilación *JIT*, establecemos modificaciones a su modelo computacional para incorporar las características de reflexión estructural ofrecidas por los lenguajes dinámicos, evaluando la mejora del rendimiento en tiempo de ejecución. El principal objetivo de esta modificación es, pues, tratar de proporcionar un soporte integrado y eficiente para dichas características dentro de la máquina, de manera que sea posible emplearla para implementar lenguajes dinámicos sobre la misma, mejorando el rendimiento en tiempo de ejecución de estos lenguajes respecto a otras implementaciones existentes. Por otra parte, la modificación del sistema es en todo momento una extensión del mismo, es decir, la compatibilidad con el código heredado es mantenida en todo momento.

Para todo ello, se diseña un nuevo modelo computacional que añade al modelo orientado a objetos basado en clases, ya existente en la máquina de partida, otro modelo que utiliza principios del modelo de orientación a objetos basado en prototipos, más adecuado para soportar reflexión estructural. Este último modelo computacional utiliza las ampliaciones desarrolladas, de forma que se siga manteniendo la compatibilidad con el código heredado y al mismo tiempo se proporcione un modelo sobre el que se puedan ejecutar cualquier lenguaje dinámico.

Se diseña un conjunto de primitivas para soportar las características de reflexión estructural ofrecidas por los lenguajes dinámicos. Dichas primitivas se implementan de una forma integrada en su estructura, logrando que estas nuevas capacidades sean parte de la funcionalidad básica ofrecida por el sistema extendido. Adicionalmente se modifica la semántica de las instrucciones adecuadas del lenguaje intermedio de la máquina virtual, de forma que el compilador *JIT* las pueda procesar siguiendo el nuevo modelo. Esto permitirá generar código nativo teniendo en cuenta las primitivas de reflexión añadidas.

La idea subyacente es dotar a la máquina de soporte nativo para lenguajes dinámicos, además del soporte para lenguajes estáticos que ya pudiese tener. Ambos tipos de lenguajes se traducirán directamente a un lenguaje intermedio común, lo que permite la interacción entre distintos lenguajes dinámicos y estáticos. Una vez realizada la implementación, se analiza tanto la ganancia de rendimiento en tiempo de ejecución como el uso de memoria frente a otros sistemas dinámicos en escenarios representativos, así como el coste de dotar de una mayor flexibilidad a la máquina estática, validando así la tesis de partida.

PALABRAS CLAVE

Reflexión Estructural, Rendimiento, Eficiencia, Introspección, Máquina Abstracta, Máquina Virtual, Lenguajes Dinámicos, Compilador *JIT*, Modelo de Objetos Basado en Prototipos, *SSCLI*.

ABSTRACT

Dynamic languages give to their programs the ability to analyze and modify its structure, behaviour and environment at runtime. The main advantage offered by those languages is that they can build software that adapts better to the possible modifications of requirements during the lifetime of a particular program. They support characteristics such as meta-programming, reflection, dynamic reconfiguration and distribution.

Some dynamic languages, like *Python*, *Ruby* or *Dylan*, are becoming widely used to develop applications that require a high level of flexibility: adaptive and adaptable software, Web applications, game engines, etc. Those languages are commonly executed over a virtual machine using an interpreter, in order to take advantage of virtual machine benefits. However, this also implies that the performance of the applications developed in those languages is worse than the performance achieved with "static" languages. This performance is even decreased further by the additional operations that dynamic languages must perform in order to support their flexible characteristics.

There are research fields dedicated to increase the performance of the applications that are executed over a virtual machine, trying to make it comparable to the performance offered by those which are directly compiled to native code. Techniques like adaptive *JIT* (Just in Time) compilation offer a significant performance increase, which allowed platforms like *Java* or *C#* to be used in commercial systems. In order to improve the performance of dynamic languages, this thesis uses the same principle that was valid to improve the performance of "static" virtual machines. The majority of virtual machines developed to allow the execution of dynamic languages had been created as interpreters. In our case we use an already existing professional virtual machine with *JIT* compilation as a basis for our work, modifying it conveniently to add structural reflection features, in order to evaluate if it is possible to increase the runtime performance of dynamic languages that are executed over it. The main objective of this modification is to try to obtain an integrated and efficient support of those features inside the modified machine, so we can use them to implement dynamic languages and to increase its performance when compared with other similar implementations. This modification is in fact an extension of the virtual machine; we maintain full backwards compatibility with the original machine characteristics and legacy code.

In order to do that, a new computational model is designed to try to unify the existing one (a class-based object-oriented model) with another model that uses some principles of the prototype-based object-oriented model, which better supports structural reflection. This new model is used by our modifications, so we can both maintain the compatibility with legacy code and offer a new computational model which can be used to execute the dynamic languages that could be added to the modified machine.

Additionally, we design a set of primitives that allows the modified system to support the structural reflection features of dynamic languages inside the currently existing languages. Any programmer will be able to use them without modifying the syntax of those languages. Those primitives are implemented in an integrated and efficient (native) form, so that they are part of the set of standard features offered by the machine. Finally, we modify the semantics of some instructions of the intermediate language that the virtual machine generates, but not the own language. This way the *JIT* compiler processes those instructions following the new computational model, allowing the system to generate native code using the new structural reflection features transparently.

With the idea described above, we give the virtual machine a native support for

dynamic languages, in addition to the support for static languages that it already has. Both types of languages are directly translated to a common intermediate language (already existing in the original machine), which allows that any language L (either dynamic or static), supported by the extended virtual machine, could interoperate with any other language L2. Additionally, maintaining the same already existing intermediate language enables us to offer full backwards compatibility with legacy code, so any program designed for the unmodified machine could be executed and return the same results.

Once the implementation is finished, we validate the performance gains and memory usage of our system, comparing it with other dynamic systems of similar characteristics over various representative scenarios, testing also the cost of incorporating more flexibility to the original system, validating the idea that is described in this thesis.

KEYWORDS

Structural Reflection, Performance, Efficiency, Introspection, Abstract Machine, Virtual Machine, Dynamic Languages, *JIT* Compiler, Prototype-Oriented Object Model, *SSCLI*.

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

SECCIÓN A: INTRODUCCIÓN

1 INTRODUCCIÓN

En éste capítulo se describen los principales objetivos buscados en el desarrollo de esta tesis doctoral, estableciendo unos aspectos generales a cumplir. Posteriormente se presenta la organización de la memoria, estructurada tanto en secciones como en capítulos.

1.1 INTRODUCCIÓN

Cuando se aborda el diseño de un proyecto *software*, se puede plantear el problema de seleccionar qué lenguaje será el más indicado para implementarlo. Aunque este problema puede abordarse teniendo en cuenta diferentes factores, uno de ellos es determinar si es adecuado para las necesidades del desarrollo que el lenguaje escogido sea dinámico. Estos lenguajes son más adecuados para aquellas aplicaciones que necesiten de un alto grado de flexibilidad, ya que ésta podría resultar más difícil de obtener con lenguajes "tradicionales" como *C++* o *Java* (denominados "estáticos"). El tipo de *software* a realizar, el entorno en el que va a operar, las características más prioritarias de la aplicación final o cómo se va a realizar dicho diseño jugarán un papel muy importante a la hora de escoger el tipo de lenguaje más apropiado. Por ejemplo, puede ser mucho más importante que la aplicación final tenga un rendimiento en tiempo de ejecución lo más óptimo posible frente a otras consideraciones como la productividad (el tiempo tardado en desarrollar el código de la misma) o bien que se prime justamente lo contrario [Ferg06]. En definitiva, el peso que los desarrolladores le quieran dar a los diferentes factores expuestos hará que la elección se incline por un tipo u otro de lenguajes.

Enlazando con lo dicho anteriormente, uno de los casos donde las ventajas de usar un lenguaje dinámico se hace patente ocurre cuando, por diversos motivos, el diseño original de una aplicación se ve sometido a múltiples cambios que pueden tener una entidad considerable. Los requisitos originales y las reglas de negocio pueden cambiar sustancialmente durante todo el desarrollo, de forma que el diseño original del *software* se vea sometido a procesos de adaptación, que podrían llegar a ser costosos de plasmar en la implementación. Una aplicación desarrollada con un lenguaje dinámico, a través de la flexibilidad que proporciona, puede responder mejor ante estos cambios.

Por tanto, son este tipo de problemas (entre otros) los pueden ser respondidos de una forma más adecuada por lenguajes dinámicos, que permiten dotar de una flexibilidad mayor al *software* desarrollado con ellos, y que de esta forma pueda adaptarse de una forma más efectiva a los diversos cambios que surjan a lo largo de su vida útil, incluso haciendo ésta adaptación en tiempo de ejecución. No obstante, como ya se ha mencionado, incluso en aquellas situaciones en las que las capacidades de los lenguajes dinámicos ofrezcan a priori múltiples ventajas, estos lenguajes pueden no seleccionarse para un desarrollo. De todos los motivos que pueden llevar a tomar esta decisión, probablemente el más importante sea el menor rendimiento en tiempo de ejecución que las aplicaciones desarrolladas con estos lenguajes normalmente poseen. Esto está motivado en gran parte por su naturaleza interpretada y por la necesidad de efectuar operaciones adicionales en tiempo de ejecución (que los lenguajes estáticos no

necesitan, por ejemplo, inferencia y comprobación de tipos), siendo ambos factores que penalizan el rendimiento de forma significativa y que, por tanto, hagan a la aplicación realizada con este tipo de lenguajes inadecuada para determinados entornos en los que se espera un rendimiento determinado.

Por otra parte, podemos constatar que hoy en día existen soluciones basadas en máquinas virtuales como plataformas de desarrollo que han incrementado significativamente su popularidad. Lenguajes como *Java* o *C#* funcionan sobre una máquina virtual, y como tales son compilados a un código intermedio de una máquina físicamente inexistente (de ahí el término "virtual"), siendo posteriormente ejecutado dicho código intermedio mediante diversas técnicas, de forma que su ejecución sobre la plataforma final de destino sea más eficiente que mediante una interpretación pura. Las investigaciones llevadas a cabo para mejorar el rendimiento del código que se ejecuta bajo una máquina virtual y las ventajas ofrecidas por este tipo de plataformas, como su elevada portabilidad, han hecho posible su empleo para el desarrollo de soluciones en un número creciente y diverso de campos. Es común también encontrar implementaciones de lenguajes dinámicos que emplean máquinas virtuales, funcionando éstas como intérpretes por ser una vía más sencilla de implementar todas sus características, pero con el mismo problema de falta de rendimiento descrito.

Esta tesis tratará pues de emplear un sistema basado en una máquina virtual y una serie de técnicas de optimización de rendimiento asociadas (como la compilación *JIT*), para comprobar si es posible aumentar el rendimiento en tiempo de ejecución de lenguajes dinámicos que se ejecuten sobre la misma, una vez sea convenientemente modificada. Para ello, partiremos de una máquina virtual profesional estática (aquella que está diseñada, implementada y optimizada teniendo en cuenta la ejecución de lenguajes estáticos, maximizando el rendimiento de su ejecución) dotada de las técnicas de optimización mencionadas, a la que modificaremos internamente su modelo computacional y estructuras, de manera que sea capaz de soportar una serie de primitivas de reflexión de lenguajes dinámicos de forma nativa. Se pretende que de esta forma sea posible implementar sobre la misma un conjunto de lenguajes dinámicos de alto nivel que usen estas primitivas, y que por tanto la máquina sea capaz de ofrecer un soporte nativo para estos lenguajes de la misma forma que ya lo ofrece para el conjunto de lenguajes "estáticos", permitiendo incluso la interoperabilidad entre ellos.

Por supuesto, esta modificación no debe alterar el funcionamiento de la máquina para ningún lenguaje ya soportado por la misma, de manera que cualquier programa que fuera válido en la máquina original siga siéndolo en la máquina extendida, devolviendo los mismos resultados ante idénticos parámetros de entrada. Por tanto, el modelo computacional de la máquina debe ser ampliado (y no sustituido) para que ambos tipos de lenguajes puedan coexistir adecuadamente (habida cuenta de que no podemos usar el existente para ambos tipos), respetando además todas aquellas propiedades adicionales que el sistema original poseyese.

A lo largo de esta tesis se describirán las diferentes alternativas existentes en cuanto a máquinas virtuales potencialmente utilizables para el trabajo planteado, las primitivas relativas a la flexibilidad ofrecida por lenguajes dinámicos que pueden implementarse, las modificaciones que se deberán hacer al modelo computacional y a otras partes de la máquina y las técnicas de optimización de aplicaciones en tiempo de ejecución aplicables, seleccionando la máquina más adecuada como punto de partida y aquellos elementos que nos resulten más convenientes para completar el trabajo descrito, mostrando los resultados obtenidos por la máquina extendida y analizando su rendimiento y eficiencia ante diferentes escenarios de manera que justifique la validez de la idea presentada.

1.2 OBJETIVOS

A continuación describiremos los objetivos principales a obtener en esta tesis de forma general, estableciendo así una serie de premisas que deberán seguirse durante el desarrollo de todo el trabajo.

1.2.1 Creación de una Máquina Virtual que Soporte Primitivas Reflectivas

Uno de los principales propósitos del trabajo a desarrollar es la construcción de una máquina virtual que tenga soporte nativo para un determinado grado de flexibilidad, proporcionado a través de la incorporación de una serie de primitivas reflectivas a la misma. La máquina no se construirá desde cero, sino que se empleará una máquina estática ya existente para su elaboración que tenga un buen rendimiento y un conjunto de técnicas de optimización de aplicaciones en tiempo de ejecución, como la compilación bajo demanda y la optimización adaptativa, para poder aplicarlas a las nuevas primitivas desarrolladas.

1.2.2 Implementación de Capacidades Dinámicas a un Coste Razonable

El grado de flexibilidad incorporado a la máquina estará condicionado por el coste en ejecución que suponga. Dado que modificar una máquina estática para dotarla de un soporte nativo para una mayor flexibilidad sin duda implicará una penalización en su rendimiento, motivado por las operaciones adicionales que deberán hacerse, se determinarán qué primitivas se van a añadir de forma que se pueda lograr un doble objetivo:

- Aumentar el grado de flexibilidad hasta proporcionar un soporte nativo adecuado para lenguajes dinámicos, equivalente al que ya poseen la mayoría de los más usados.
- Penalizar el rendimiento del sistema dentro de unos límites aceptables.

Por tanto, otro de los objetivos de este trabajo es determinar el conjunto de operaciones adecuado para alcanzar los dos objetivos vistos, logrando un balance flexibilidad/rendimiento lo más óptimo posible.

1.2.3 Desarrollo de un Modelo Computacional Reflectivo

Dado que una de las principales razones por las que se dota a la máquina de más flexibilidad es la de implementar sobre ella lenguajes dinámicos, deberán hacerse cambios significativos al modelo computacional de la misma para poder darles un soporte adecuado. Dado que la máquina de partida estará preparada únicamente para ofrecer soporte para lenguajes estáticos, soporte que por otra parte se debe conservar en los mismos términos en los que ya existe, se deberá pues ampliar el modelo computacional de la misma para proporcionar también un soporte equivalente para lenguajes dinámicos, de manera que ambos tipos de lenguajes puedan "convivir" e interactuar entre sí.

Por tanto, otro de los principales objetivos a lograr en esta tesis es obtener un modelo computacional que permita soportar ambos tipos de lenguajes, sin perder las características ya existentes ni permitir incoherencias en dicho modelo para cualquiera de ellos al hacer determinadas operaciones.

1.2.4 Incrementar el Rendimiento de Primitivas Reflectivas

Por último, pero no por ello menos importante, el principal objetivo de los cambios que se realizarán es obtener un mejor rendimiento con nuestro sistema extendido que con otras implementaciones de lenguajes dinámicos ya existentes. Esto se puede contemplar desde tres puntos de vista:

- Lograr un mejor rendimiento que los lenguajes dinámicos ante programas que usen primitivas reflectivas en su código, usando pues características de flexibilidad: **prueba de código reflectivo**.
- Conseguir perder un rendimiento mínimo respecto a la máquina original cuando se ejecutan programas "estáticos" (únicos para los que el sistema original está diseñado, al no contar con soporte nativo para un alto grado de flexibilidad, que será lo que le añadiremos): **coste del cambio**.
- Conseguir que el sistema extendido tenga mejor rendimiento que un lenguaje dinámico cuando ambos ejecutan un mismo programa que no haga uso de capacidades reflectivas. La intención de esta medida es precisamente comprobar si el rendimiento de nuestro sistema es mejor que el de un lenguaje dinámico con código que no haga uso de características de flexibilidad, para así comprobar cómo se comporta ante ambas clases de código: **prueba de código no reflectivo**.

Por ello, el objetivo último de esta tesis es lograr mejorar globalmente el rendimiento de los lenguajes dinámicos implementados sobre nuestro sistema a un coste razonable, mostrándose el trabajo realizado como una posible vía para acercar el rendimiento de los lenguajes dinámicos al de los estáticos y actuando por tanto directamente sobre su principal inconveniente.

1.3 ORGANIZACIÓN DE LA TESIS

A continuación se presenta la estructura de la tesis, agrupando los capítulos en secciones con un contenido acorde.

1.3.1 Introducción y Características del Sistema

En esta sección se narrará la introducción, objetivos generales y organización de la tesis. Otra parte de la misma consistirá en establecer el conjunto de requisitos impuestos al sistema, que serán utilizados principalmente para tres tareas:

- Para evaluar las aportaciones y carencias de los sistemas estudiados en las siguientes secciones frente a las necesidades planteadas, teniendo en cuenta cada uno de los elementos que debemos considerar de cara a desarrollar el objetivo final y poder así tomar las decisiones correctas en cuanto a los sistemas usados y las características implementadas.
- Para fijar la arquitectura global del sistema, optimizaciones a desarrollar y restricciones en cuanto a lo que se puede y no se puede modificar.
- Para evaluar los resultados del sistema propuesto, comparándolos con otros sistemas similares existentes estudiados, validando la tesis expuesta.

1.3.2 Estudio de Sistemas Existentes

En esta sección se lleva a cabo un estudio general de los sistemas similares al buscado, así como de todo lo relacionado con los diferentes conceptos que se van a estudiar en ésta tesis. En el capítulo 3 se introducen los conceptos de reflexión y una selección de sistemas que la usan y sus aplicaciones, mientras que en el capítulo 4 se introducirán conceptos teóricos acerca de los sistemas basados en prototipos, que se usarán para implementar el modelo reflectivo dentro del sistema base. Posteriormente, en el capítulo 5 se estudiarán los lenguajes dinámicos, para tener una panorámica clara de las funcionalidades que se pretenden incorporar en esta tesis.

El capítulo 6 presenta el concepto de máquina abstracta y de máquina virtual, así como las ventajas generales de su utilización. También se incluye un estudio y evaluación de distintos tipos de máquinas abstractas existentes actualmente.

En el capítulo 7 se hablará de las diferentes técnicas que actualmente son usadas para la optimización de sistemas en tiempo de ejecución, además de hacer un estudio más en profundidad de una de estas tecnologías: la tecnología de compilación *Just-In-Time* (*JIT*).

1.3.3 Diseño e Implementación del Sistema

En esta sección se introduce el sistema propuesto, basándose en los requisitos impuestos y los sistemas estudiados. La arquitectura global del sistema se presenta en el capítulo 8, mientras que el diseño general del motor computacional se describe en el capítulo 9. Los cambios realizados al modelo computacional del sistema base se presentan en el capítulo 10 y en los dos capítulos siguientes se presentan dos aspectos muy importantes en el diseño del mismo: La interoperabilidad y la optimización dinámica del modelo, por ese orden. Finalmente, la implementación del prototipo de pruebas se describe en el capítulo 13.

1.3.4 Aplicaciones, Evaluación, Conclusiones y Trabajo Futuro

La evaluación del sistema presentado en esta tesis, comparándolo con otros existentes, se llevará a cabo en el capítulo 14, teniendo muy presentes los requisitos establecidos al comienzo de ésta para realizar dicha evaluación.

Un conjunto de posibles aplicaciones prácticas del sistema desarrollado se presenta en el capítulo 15, presentando tanto aplicaciones existentes desarrolladas como posibles ideas y conceptos de viabilidad probada.

Finalmente, a partir del capítulo 16, se muestran las conclusiones globales de la tesis, las futuras líneas de investigación y el trabajo futuro a realizar (capítulo 17).

1.3.5 Apéndices

En el apéndice A se presenta una descripción técnica del sistema base usado para las modificaciones (*SSCLI*), abarcando los elementos más significativos de su arquitectura y los conceptos más importantes involucrados en su construcción.

En el apéndice B se presenta la especificación del conjunto de primitivas reflectivas desarrolladas, mientras que el apéndice C recopila las tablas de datos usadas en el apartado de evaluación del sistema. Finalmente, el apéndice D presenta el conjunto de referencias bibliográficas utilizadas en este documento.

2 REQUISITOS DEL SISTEMA

En este capítulo estableceremos todos aquellos requisitos que el trabajo a desarrollar debe cumplir, agrupándolos en diferentes ámbitos en función de a lo que se refieran.

2.1 REQUISITOS ARQUITECTÓNICOS

Bajo este punto se agruparán los requisitos que exigiremos al sistema base sobre el que se realizarán las modificaciones que desarrollen los conceptos presentados en esta tesis. Estos requisitos servirán principalmente como elemento para discriminar qué sistemas se van a tener en consideración y cuales son descartables, ayudando por tanto a seleccionar el sistema final. A continuación se hará pues un desglose pormenorizado de estos requisitos:

2.1.1 Máquina Virtual

Partiendo del hecho de que el sistema escogido deberá estar basado en una máquina virtual modificable, que se pueda emplear para implementar todas las primitivas necesarias para el trabajo a desarrollar en esta tesis, la máquina a escoger deberá ser estable y ofrecer posibilidades reales de ampliación y modificación. Estas posibilidades contemplarían:

- **La capacidad de poder modificar convenientemente todo el código** de cualquier módulo de la misma, e incorporar esos cambios fácilmente. Idealmente, no deberían existir ningún tipo de restricciones para poder modificar cualquier parte del mismo, lo que incluye:

    - Acceso completo a las bibliotecas estándar de objetos ofrecidas por el sistema, y capacidad para modificar tanto las interfaces como la funcionalidad de cualquier objeto o clase ya existente. Además, debe ser posible añadir nuevos objetos y funcionalidades que se integren en el sistema de la misma forma que los que ya están presentes. Por último, deberán proporcionarse herramientas de validación y compilación del nuevo código añadido, lo suficientemente potentes para procesar estos cambios comprobando que no se alteran inadecuadamente otras partes del sistema en la medida de lo posible.
    - Acceso al código de la máquina virtual del sistema y, en general, a cualquier módulo del mismo. Se debe pues permitir el estudio, modificación o ampliación de cualquiera de los módulos "internos" del

sistema, entendiendo por ello a cualquier código fuente que forme parte de la implementación de sus características. Esto requerirá también de herramientas adecuadas para incorporar dichos cambios de forma eficiente (sin necesidad de volver a construir siempre el sistema completo en cada cambio) y correcta, que sean capaces de detectar, también en la medida de lo posible, las incompatibilidades en las que podamos incurrir al incorporar dichas modificaciones.

  - Acceso a cualquier analizador, compilador e intérprete que el sistema base posea, para poder modificar, si fuese necesario, cualquier aspecto relativo a cómo se procesan las sentencias de un lenguaje cualquiera soportado por el sistema, incluyendo el lenguaje intermedio que el mismo genera.
  - *Acceso a las gramáticas y especificación de las instrucciones de cualquier lenguaje d*el sistema, incluyendo el lenguaje intermedio generado.

- **La no existencia de limitaciones al respecto a nivel legal**, con una licencia adecuada para hacer toda clase de modificaciones que puedan ser necesarias. Por ello, será necesario que el sistema permita el acceso y modificación de su código sin incurrir en problemas legales ni limitaciones de ningún tipo, salvo aquéllas que sean razonables por motivos de *copyright* de algún módulo o similar, pero que en ningún caso afecten a partes que sea necesario modificar para este trabajo de manera que se limite el campo de actuación que tengamos al respecto. Esto aconseja el empleo de sistemas de código abierto (*Open Source*) o similares.
- **Posibilidades de consulta de una documentación suficiente** para facilitar la labor de modificación y ayudar a realizar los cambios de forma más coherente, tratando de minimizar el número de errores o incompatibilidades en las que podamos incurrir accidentalmente. Se deberá pues contar con una documentación lo más extensa y detallada posible, que permita abordar las modificaciones a realizar con un conocimiento del sistema elevado, y poder saber aproximadamente cómo reaccionará cada parte del sistema a los cambios que se le introduzcan. Por tanto, la documentación que el sistema debería poseer no debe limitarse a describir la estructura de los elementos, sino también su funcionamiento y comportamiento ante determinadas situaciones. Dicha documentación debe abarcar tanto su arquitectura interna completa como los lenguajes soportados y sus librerías de clases, por citar los elementos más importantes.

Además, esta máquina deberá poseer un rendimiento aceptable de base y algún tipo de técnica de optimización en tiempo de ejecución efectiva de las aplicaciones que sobre ella se ejecuten, adecuada a los fines que se persiguen. Por otra parte, la versión utilizada debe ser lo suficientemente estable y desarrollada como para poder emplearla sin la necesidad de actualizar la misma cada poco tiempo, dado que las modificaciones que se lleven a cabo en nuevas versiones podrían interferir con las desarrolladas en esta tesis y obligarnos a repetir o adaptar el trabajo ya realizado.

2.1.2 Independencia del Lenguaje de Programación

Dentro del abanico de sistemas basados en máquinas virtuales que podemos escoger como punto de partida, podemos establecer una división entre aquéllos que soportan un único lenguaje y aquéllos que proporcionan soporte para varios. Idealmente,

el sistema que finalmente sea escogido debería proporcionar soporte para varios lenguajes, y las modificaciones que en él hagamos deberían a su vez ser independientes de cualquier lenguaje que la máquina posea, de forma que todos ellos puedan acceder y usar convenientemente cualquier capacidad nueva añadida.

Por otra parte, el requisito de soportar múltiples lenguajes deberá estar acompañado por la posibilidad de desarrollar nuevos lenguajes para dicha plataforma, que también sean capaces de acceder, al igual que los existentes, a todas las características nuevas que se añadan. No obstante, lo más interesante de esta característica es la posibilidad de añadir lenguajes dinámicos al sistema de partida que convivan con los ya existentes, usando las nuevas primitivas para su implementación.

2.1.3 Independencia del Problema

El sistema de partida no debe estar orientado a la resolución de un problema o conjunto de problemas concretos, sino que debe ser de propósito general, lo que le permitirá adaptarse a la gran mayoría de desarrollos que se puedan plantear y crear una solución adecuada para ellos. El sistema no deberá pues tener una única utilidad específica, como pueden ser aspectos relativos a seguridad o facilitar la migración de *software*, por ejemplo.

2.1.4 Soporte para Capacidades Reflectivas

Si bien el trabajo a desarrollar en esta tesis tratará de ampliar las capacidades reflectivas a un sistema concreto, un sistema de base que cuente ya con ciertas capacidades en este sentido (introspección, programación generativa,…) puede proporcionar algunas ventajas que no pueden ser desdeñadas:

- La técnica empleada por el sistema para acceder a las capacidades reflectivas que ya posee puede ser empleada por las nuevas capacidades introducidas para que el usuario pueda también acceder a ellas, logrando una mejor integración.
- Facilitarían el desarrollo, ya que todo aquello que el sistema ya posea no deberemos implementarlo y podemos usar esa parte ya existente como base para construir sobre la misma las nuevas capacidades.

Por ello, un factor a tener muy cuenta será el hecho de que el sistema escogido ya posea algún tipo de capacidad reflectiva como las descritas y por tanto un cierto grado de flexibilidad que pueda servir como punto de partida para su ampliación.

2.1.5 Independencia de la Plataforma y Portabilidad

El sistema base escogido deberá tener una implementación operativa en el mayor

número de plataformas posible. Además, cualquiera de estos sistemas portados debe proporcionar la misma funcionalidad que el "original", de manera que no existan programas que se puedan ejecutar en unas plataformas y en otras no (al no ser implementaciones completas).

La independencia de la plataforma es una de las principales ventajas que pueden obtener los programas por el uso de una máquina virtual para su implementación. El código de cualquier lenguaje soportado por la misma se podría traducir a un lenguaje intermedio independiente de la plataforma, que luego puede ser nuevamente traducido a otro que sea nativo de la plataforma escogida, traduciendo de esta forma un solo lenguaje a código nativo en lugar de los N que soporte la máquina. Cualquier lenguaje soportado por la misma (incluyendo los lenguajes dinámicos que se puedan implementar sobre la máquina extendida) debe contar con esa posibilidad, e interesa que el sistema extendido permita a los nuevos lenguajes adquirirla fácilmente.

2.1.6 Proyección del Sistema para el Desarrollo de Aplicaciones Reales

La elección del sistema base debería recaer sobre una máquina virtual existente y que preferiblemente tenga una comunidad amplia de usuarios en activo y/o aplicaciones para proyectos reales o en empresas. También se contemplará la posibilidad de que dichas aplicaciones puedan ser realmente desarrolladas en el futuro o que las modificaciones realizadas puedan portarse a un sistema más acorde a estas características. El objetivo es lograr que el conjunto de usuarios del sistema base sea lo más elevado posible ya desde el principio, que se pueda emplear de la misma forma el sistema extendido y sobre todo que se puedan aprovechar sus nuevas capacidades para desarrollar soluciones a los mismos problemas, teniendo en cuenta la flexibilidad incorporada al mismo.

2.2 REQUISITOS DE COMPATIBILIDAD

El sistema base debe ser ampliado sin alterar o limitar sus capacidades ya existentes. Por tanto, cualquier modificación al sistema debe hacerse con especial cuidado para no alterar ninguna de las funcionalidades ya poseídas, integrándose las nuevas funcionalidades en el mismo de forma lógica y coherente. Este requisito contempla los siguientes aspectos:

2.2.1 Conservar las Características del Sistema

Toda característica ya poseída por el sistema base, como portabilidad, interoperabilidad, etc. deberá ser poseída también por su ampliación. Por ejemplo, si el sistema tenía capacidad para ser portado a otras plataformas, la nueva funcionalidad no debe alterar dicha capacidad. Las modificaciones planteadas no deberán por tanto alterar ninguna de estas características y nada de lo que se implemente se hará limitando o

perdiendo cualquiera de ellas, alterando partes vitales del sistema de manera que ya no puedan portarse.

2.2.2 Compatibilidad hacia Atrás

Es de vital importancia asegurarse de que cualquier programa válido en el sistema original lo siga siendo en el sistema extendido, es decir, los resultados ofrecidos por cualquier programa ante una determinada entrada no deben verse alterados en el nuevo sistema.

2.2.3 Ampliación del Modelo Computacional para Soportar Lenguajes Dinámicos

El modelo computacional del sistema original debe ampliarse de manera que, manteniendo las mismas capacidades que ya tenía, ahora sea capaz de soportar la implementación de lenguajes dinámicos sobre él mismo, sin que el modelo resulte incoherente para ningún tipo de lenguaje. Esto exigirá el estudio de un modelo adecuado para satisfacer las necesidades tanto de los lenguajes soportados ya por la máquina seleccionada (probablemente sólo estáticos) como los dinámicos, que pueda ser empleado alternativamente para la implementación sobre la máquina de nuevos lenguajes de cualquiera de estos dos tipos sin violar ninguna restricción.

2.2.4 Interoperabilidad Ampliada de Lenguajes

En el caso de que el sistema escogido soporte varios lenguajes, sería muy interesente poder acceder desde un lenguaje dado a datos y operaciones realizadas en cualquiera de los otros lenguajes soportados por la máquina, permitiendo la convivencia de código realizado en múltiples lenguajes. Esto implica, por ejemplo, que cualquier conjunto de clases (con sus atributos y métodos) creado con uno de los lenguajes pueda ser usado sin restricciones por cualquiera de los otros. Este requisito permitirá que, en caso de proporcionar acceso a las nuevas capacidades mediante la creación de una interfaz de operaciones de alto nivel, cualquier lenguaje de la plataforma escogida pueda usar estas operaciones sin necesidad de repetir su implementación para cada lenguaje al que queramos añadirlas.

Además, el sistema extendido no sólo tendrá que conservar esta característica (requisito establecido anteriormente), sino que también se deberá aplicar a los nuevos lenguajes que se le puedan añadir a la máquina, sean o no dinámicos. El sistema extendido final deberá soportar ambos tipos de lenguajes, y permitir que el código desarrollado en un lenguaje de uno de los tipos pueda ser empleado coherentemente por cualquier otro lenguaje del otro tipo sin restricciones. Por tanto, todo lenguaje dinámico que se le pueda añadir al sistema extendido deberá tener las mismas posibilidades de interactuación que cualquier otro existente previamente.

2.3 REQUISITOS DE REFLEXIÓN

En esta sección se describirán aquellos aspectos generales que se deben lograr con la ampliación de la plataforma base. Los requisitos de reflexión describen el nivel de flexibilidad que el sistema ampliado va a tener y qué se pretende lograr con dicho nivel (capacidades que el sistema poseerá). Estos requisitos son el conjunto de funcionalidades deseado, pero a partir de los mismos es posible que se estudie la ampliación de alguno de ellos, siempre y cuando el coste impuesto por la incorporación de esta ampliación sea razonable, teniendo en cuenta qué ganancia de flexibilidad es aportada por dichas ampliaciones y qué impacto de rendimiento tiene su incorporación.

2.3.1 Introspección

Una de los servicios básicos del sistema a desarrollar es que toda aplicación pueda conocer el entorno en el que se está ejecutando. Mediante introspección, una aplicación podrá por ejemplo conocer si todas las operaciones y elementos que necesita están o no presentes y actuar en consecuencia. Cualquier aplicación deberá pues ser capaz de conocer el estado de cualquier parte del sistema en tiempo de ejecución. Gracias a esta característica será posible desarrollar aplicaciones dinámicamente adaptables a contextos, puesto que podrán analizarse éstos en tiempo de ejecución [Ortin01].

Esta característica deberá estar presente en el sistema obligatoriamente, ya que proporciona un nivel básico de flexibilidad sin el cual no se pueden hacer mejoras en este sentido.

2.3.2 Modificación Dinámica de la Estructura del Sistema

El concepto de introspección introducido en el punto anterior permite consultar el estado del sistema, pero no capacita al programa para modificar ningún aspecto del mismo. Para solventar esta carencia, se deberá implementar la capacidad de modificar partes del mismo en tiempo de ejecución, añadiendo, modificando o eliminando atributos y métodos de las clases o instancias según sea necesario.

El objetivo de este requisito es que el sistema pueda dar un mejor soporte para lenguajes dinámicos, creando un soporte nativo para dotar al sistema de capacidades que permitan la creación de programas cuyos requisitos y/o reglas de negocio puedan cambiar dinámicamente durante su ejecución, y sean capaces de reaccionar adecuadamente a estos cambios sin necesidad de parar y volver a compilar dicho programa. Esto incluye:

- **Modificación de la estructura de clases y/o instancias**: Cualquier atributo de

cualquier clase del sistema (nuevas clases creadas, ya existentes antes de la ampliación del sistema base, de la librería estándar del mismo, etc.) o instancia de las mismas podrá ser modificado o eliminado. Al mismo tiempo, cualquiera de estos elementos admitirá la creación y asignación de nuevos atributos sin limitación de número, desarrollando en todo caso un modelo de funcionamiento coherente y que no de lugar a ambigüedades.

- **Modificación de métodos**: Al igual que con los atributos, cualquier método de las clases del sistema podrá ser modificado, eliminado o añadido a las mismas. Esto debería permitir, entre otras, operaciones como cambiar el código de un método determinado en tiempo de ejecución.

Por tanto, este sistema debe contar con nuevas funcionalidades que permitan dotarlo de capacidades de flexibilidad adecuadas a un coste razonable. En caso de poder implementar nuevas características que amplíen los requisitos descritos anteriormente, debe estudiarse su coste de implementación de estas características cuidadosamente para justificar su rentabilidad.

Por otra parte, las nuevas funcionalidades desarrolladas deben estar completamente integradas en el sistema base. Por tanto deberán ser construidas dentro de la jerarquía de clases ya existente, estando completamente integradas en la estructura del sistema y manteniendo un método de acceso idéntico al ya existente para otras funcionalidades, empleando en la medida de lo posible los medios de los que el sistema ya dispone para su implementación. Esto permitirá ofrecerlas como un servicio del sistema más, logrando que los lenguajes que se incorporen al sistema basados en estas nuevas funcionalidades puedan comunicarse con él de la misma forma que los ya existentes, y que por tanto estos puedan aprovecharse de todas las características del mismo y colaborar con otros lenguajes de una forma transparente.

2.4 REQUISITOS DE EFICIENCIA

Para comprobar si el rendimiento ofrecido por la implementación de las nuevas funcionalidades cumple con los objetivos establecidos en esta tesis para el sistema extendido, estableceremos unas medidas de rendimiento concretas, comparando el sistema extendido con otros sistemas y estableciendo hasta qué punto podemos aceptar una pérdida de rendimiento o cuál es la ganancia esperada según el caso. Otro factor a tener en cuenta es el uso adicional de memoria requerido para las posibles ganancias de rendimiento que obtengamos, por lo que finalmente los requisitos harán mención a la eficiencia del sistema (rendimiento frente a coste de memoria).

2.4.1 Rendimiento Frente a Sistemas de Características Similares

El rendimiento del sistema ampliado debe ser superior al de otros sistemas cuyas propiedades sean similares y que soporten lenguajes dinámicos, en aquellas aplicaciones que hagan uso de características de reflexión estructural, ejecutándose éstas sobre ambos sistemas de forma equivalente.

Para cumplir este objetivo de una forma más sencilla, deberán estudiarse un conjunto de los lenguajes dinámicos existentes en el mercado actual más representativos, y escoger uno cuyas características le hagan superior a los demás para usarlo como lenguaje de pruebas, empleando para ello los requisitos establecidos en esta tesis, dado que tanto el lenguaje escogido como el sistema extendido tienen un ámbito de aplicación similar. Posteriormente, se deberán seleccionar las versiones o implementaciones de este lenguaje dinámico más representativas en cuanto a uso y funcionalidad, para efectuar las comparaciones más fiables posibles. De esta forma, no tendremos que comparar el sistema con multitud de plataformas alternativas, sino con aquella más afín al nuestro sistema extendido según sus características, superior a las demás según los objetivos de esta tesis, y que al mismo tiempo demuestre tener un uso extendido en el mercado actual y un rendimiento aceptable.

2.4.2 Penalización de Rendimiento Respecto al Sistema de Partida

Dado que al aumentar el grado de flexibilidad ofrecido por el sistema es muy probable obtener un rendimiento inferior respecto al original, debido a que existirá un mayor número de operaciones a realizar en tiempo de ejecución que son necesarias para soportar adecuadamente dicho grado de flexibilidad, es necesario pues limitar la magnitud de esta penalización. Se debe minimizar este efecto procurando, por ejemplo, disminuir todo lo posible el número de operaciones adicionales a realizar, o que éstas sólo tengan efecto cuando realmente se pueda asegurar que existen indicios de que puedan obtener algún resultado relevante (por ejemplo, minimizando el número de operaciones extra a realizar si el sistema no emplea ninguna de las nuevas capacidades sobre una determinada entidad).

Por tanto, la diferencia de rendimiento en este caso debe ser lo más pequeña posible cuando la aplicación a ejecutar no use las nuevas características que dotan al sistema de más flexibilidad, obteniendo unos resultados de rendimiento lo más similares posible a los del sistema original.

2.4.3 Eficiencia de las Optimizaciones Realizadas

Toda ganancia de rendimiento que se obtenga en el sistema debe hacerse con un coste de memoria razonable, es decir, que no nos servirán aquellas ganancias de rendimiento cuyo coste de memoria se juzgue desproporcionado respecto a la magnitud de esta ganancia. El diseño de las primitivas de reflexión deberá pues tener muy en cuenta este requisito de cara a decidir las soluciones arquitectónicas empleadas para crear determinados elementos del sistema.

Por tanto, en todo momento se busca una relación razonable entre el rendimiento obtenido por una determinada operación y la memoria adicional ocupada por el sistema extendido para implementarla (la eficiencia de la operación), justificando de esta forma la existencia de cada operación en su implementación actual.

SECCIÓN B: ANÁLISIS DE SISTEMAS RELACIONADOS

3 REFLEXIÓN

La reflexión es un medio mediante el cual los sistemas que la implementan pueden aumentar su flexibilidad. La reflexión permite a un sistema conocer aspectos de sí mismo y modificarlos, logrando así que el propio sistema sea capaz de autoexaminarse y tomar decisiones acerca de las diferentes partes en las que está dividido y de todos sus componentes. El grado o nivel de reflexión que un sistema implementa es precisamente lo que marcará hasta qué punto un sistema puede conocer y actuar sobre su estructura o comportamiento, y por tanto determina el grado de flexibilidad que finalmente posea. La reflexión puede ocurrir tanto en tiempo de ejecución como en tiempo de compilación y también puede actuar sobre diversas entidades (objetos de usuario, objetos del sistema, el propio lenguaje de programación,...), todo ello determinado por el tipo de reflexión que haya sido introducido en el sistema.

Los lenguajes dinámicos hacen uso extensivo de la reflexión para soportar precisamente un alto grado de flexibilidad, una de sus principales ventajas, y es por ello por lo que en esta tesis se va a hacer una descripción de la misma y de muchos conceptos relacionados con ella, con la intención de implementarla hasta un nivel que se juzgue razonable en función de los objetivos planteados.

El objetivo de este capítulo es pues introducir los diferentes conceptos relacionados con la reflexión y los diferentes niveles de la misma. Si bien existen diferentes criterios para clasificar la reflexión [Kirby92], se empleará como base en este capítulo la clasificación propuesta en [Ortin01], por considerarla más cercana a los objetivos concretos de esta tesis.

Además, haremos un estudio de las características de un conjunto de los sistemas reflectivos existentes actualmente según el grado de reflexión que empleen, presentando una clasificación en función de diversos criterios también preestablecidos en [Ortin01], empleando los conceptos vistos en la primera parte del capítulo.

3.1 CONCEPTOS DE REFLEXIÓN

3.1.1 Reflexión

Reflectividad o reflexión *(reflection*) (usaremos ambos términos indistintamente) es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo, así como ajustar su comportamiento en función de ciertas condiciones [Maes87].

En un sistema reflectivo, su dominio computacional (el conjunto de información que computa), estará formado por la unión de su dominio computacional convencional y su propia estructura y comportamiento. Por ello, un sistema reflectivo será capaz de acceder, analizar y modificarse a sí mismo, tratando su propia estructura y semántica de la misma forma que trataría estructuras de datos de un programa de usuario normal

cualquiera, integrando entonces en su dominio computacional su propia estructura y comportamiento bajo una forma de tratamiento homogénea.

3.1.2 Sistema Base

Se denomina sistema base de uno dado a aquel sistema de computación que es dominio de otro sistema computacional distinto, al que se denomina metasistema [Golm97]. El sistema base es el motor de computación (intérprete *software* o *hardware*) de un programa de usuario. A lo largo de esta tesis se empleará este término para dos conceptos diferentes, distinguibles fácilmente por el contexto; no debe confundirse esta definición de sistema base empleada en reflexión con las referencias al sistema base que se va a emplear en esta tesis, que se refiere al sistema que se va a usar de partida para construir sobre él las modificaciones planteadas.

3.1.3 Metasistema

Se denomina metasistema a aquel sistema computacional que posee por dominio de computación a otro sistema computacional denominado sistema base. Un metasistema es por tanto un intérprete de otro intérprete. La relación entre todos los elementos se verá claramente en la figura 3.1, presentada posteriormente.

3.1.4 Cosificación

La cosificación o concretización *(reification)* es la capacidad de un sistema para acceder al estado de computación de una aplicación, como si se tratase de un conjunto de datos propios de una aplicación de usuario [Smith82].

La cosificación define el salto del entorno de computación de usuario al entorno de computación del intérprete de dicha aplicación de usuario. Hablando en los términos que hemos definido en los dos puntos anteriores, la cosificación es por tanto la posibilidad de que un sistema base acceda a su metasistema, acceso en el que además se puede manipular el sistema base como si de datos se tratase.

Por tanto, si podemos cosificar[1] un sistema, podremos acceder y modificar su estructura y comportamiento y por lo tanto podremos añadir a su dominio computacional su propia semántica. Tal y como se ha expuesto anteriormente, esta es precisamente la definición de un sistema reflectivo.

Los conceptos mencionados hasta ahora aparecen reflejados la figura 3.1.

1 Cosificar o concretizar: transformar una representación, idea o pensamiento en cosa.

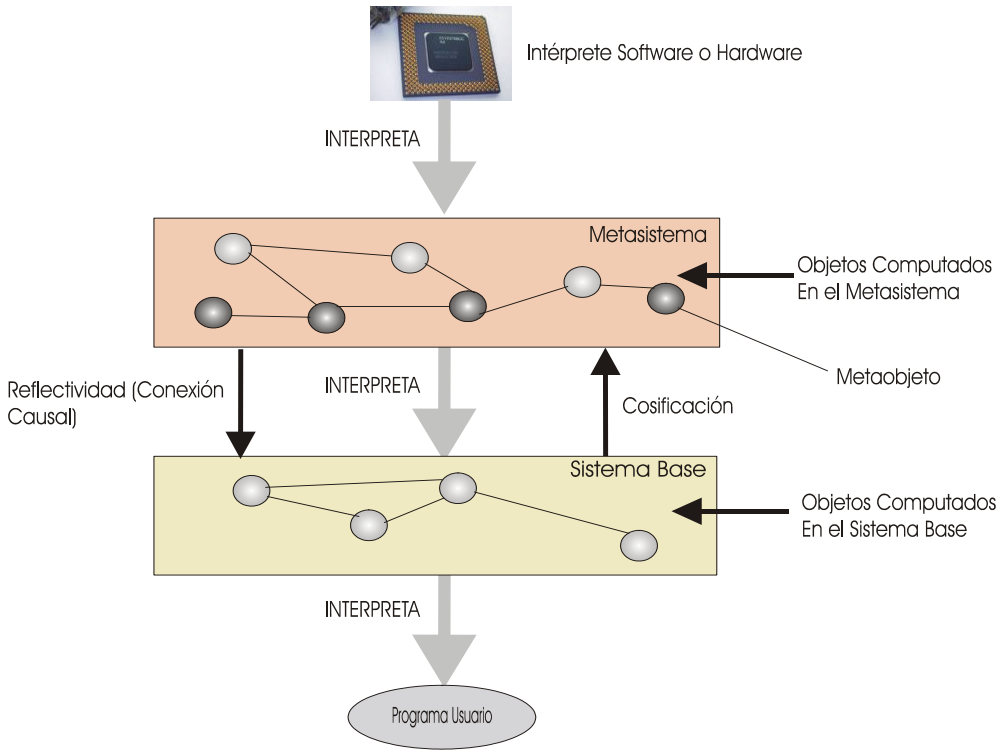

Figura 3.1: Entorno de Computación Reflectivo

3.1.5 Conexión Causal

Se dice que un sistema es causalmente conexo si permite la cosificación del sistema base, es decir que, tal y como se definió anteriormente, su dominio computacional alberga el entorno computacional de otro sistema (metasistema) y además, al modificar la representación de éste, los cambios realizados causan un efecto en la propia computación del sistema base [Maes87b].

Un sistema reflectivo es aquél que es capaz de ofrecer cosificación completa de su sistema base y de implementar un mecanismo de conexión causal. La forma en la que los distintos sistemas reflectivos desarrollan esta conexión puede variar y da lugar a un amplio abanico de soluciones de las que veremos posteriormente algunos ejemplos representativos.

3.1.6 Metaobjeto

Si se usa la orientación a objetos como paradigma de un sistema reflectivo concreto, un metaobjeto puede definirse como un objeto del metasistema que contiene información y comportamiento propio del sistema base [Kiczales91]. No obstante, la abstracción de metaobjeto no tiene porqué referirse necesariamente a un objeto concreto propio del sistema base, sino que puede representar cualquier elemento que forme parte de éste, como por ejemplo el mecanismo del paso de mensajes u otros conceptos de un nivel de abstracción mayor.

3.1.7 Reflexión Completa

Se dice que tenemos reflexión completa *(completenes*s) [Blair97] cuando se cumplen las siguientes condiciones:

- Cosificación total: La cosificación total ocurre cuando desde el metasistema es posible acceder a cualquier elemento del sistema base, sin que existan limitaciones.
- Conexión causal sin restricciones: La conexión causal se produce para cualquier elemento modificado, es decir, cualquier cambio hecho en el entorno computacional es reflejado en el sistema base.

Si se dan ambas condiciones, el metasistema ofrecerá una representación íntegra del sistema base, de manera que cualquier elemento podrá ser modificado. Como se verá en la siguiente sección de este capítulo, para que la implementación de la reflexión sea más sencilla, la mayoría de los sistemas existentes limitan a priori el conjunto de características del sistema base que pueden ser modificadas, estableciendo diferentes compromisos en el mecanismo de reflexión que implementan.

3.2 LA REFLEXIÓN COMO UNA TORRE DE INTÉRPRETES

Para explicar el concepto de reflexión computacional y para implementar prototipos que demuestren su utilidad, Smith identificó el concepto de reflexión computacional como una torre de intérpretes [Smith82].

Se diseñó una arquitectura en el que todo intérprete de un lenguaje era a su vez interpretado por otro intérprete, estableciendo de esta forma una "torre" de interpretaciones. El programa de usuario es ejecutado en un nivel de interpretación inicial (nivel 0), gracias a un intérprete que lo procesa desde el nivel computacional superior (nivel 1). En tiempo de ejecución el sistema podrá cosificarse, pasando el intérprete del nivel 1 a ser computado por otro intérprete que a su vez tendrá un nivel superior (nivel 2). De esta forma, se ve claramente cómo la reflexión computacional implica una computación de la computación. En este caso (Figura 3.2) un nuevo *intérprete'*, desde el nivel 2, pasa a computar el intérprete inicial. El dominio computacional del *intérprete'* mostrado en la Figura 3.2 estará formado por la aplicación de usuario (nivel 0) y la interpretación directa de ésta (nivel 1). Nótese que en esta situación tenemos acceso a mucha información del sistema, ya que se podrá acceder tanto a la aplicación de usuario (por ejemplo, para modificar sus objetos) como a la forma en la que ésta es interpretada (por ejemplo, para modificar la semántica del paso de mensajes). El reflejo de dichos cambios en el nivel 1 es lo que se denomina reflexión.

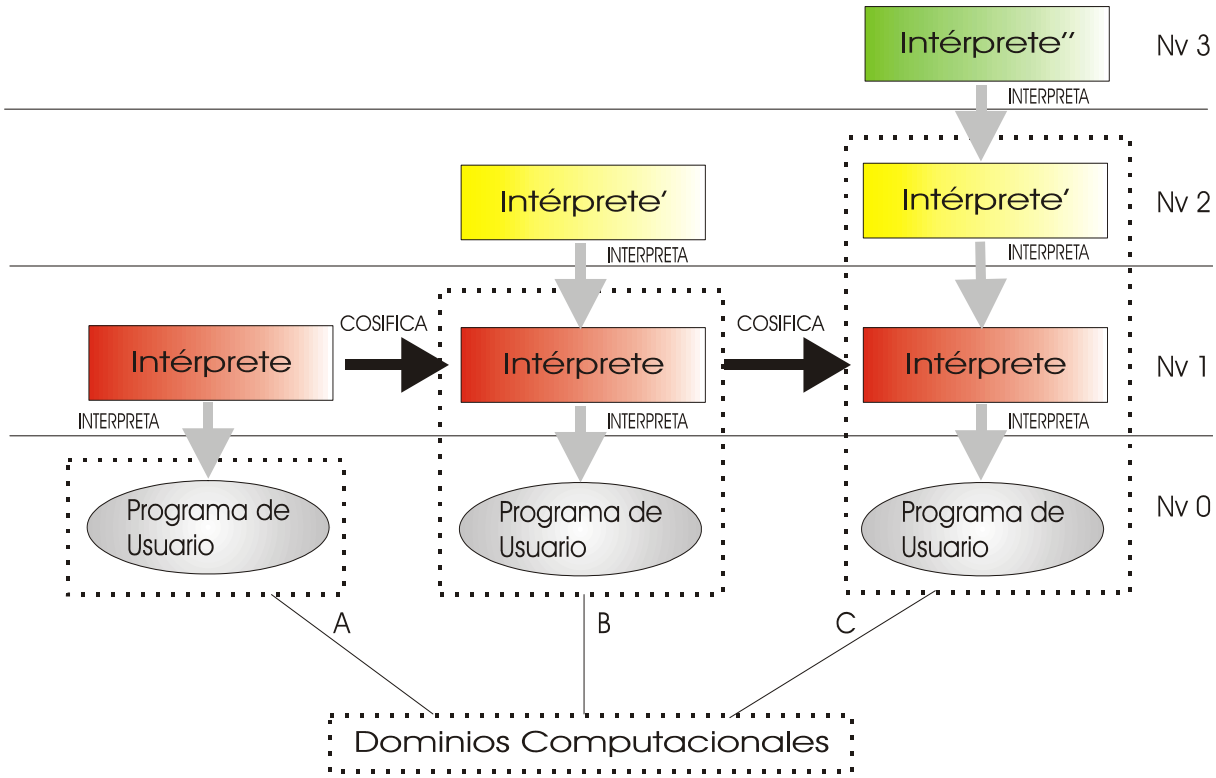

Figura 3.2: Torre de intérpretes de Smith

Con esta estructura, la cosificación se puede aplicar tantas veces como se desee. Si el *intérprete'* procesa al intérprete del nivel 1, es posible cosificar este contexto para obtener un nuevo nivel de computación (nivel 3). De esta forma se crearía un nuevo *intérprete''* que procesase los niveles 0, 1 y 2, ubicándose éste por tanto en el nivel computacional 3. En este caso podríamos reflejar la forma en la que se refleja la aplicación de usuario (tendríamos una meta-meta-computación).

Un ejemplo clásico es la depuración de un sistema *(debug*). Si estamos ejecutando un sistema y deseamos depurar mediante una traza todo su conjunto, podremos cosificar éste, para generar información relativa a su ejecución desde su intérprete. Todos los pasos de su interpretación podrán ser trazados. Si deseamos depurar el intérprete de la aplicación en lugar de la propia aplicación, podremos establecer una nueva cosificación aplicando el mismo sistema [Douence99].

Este entorno definió el concepto de reflexión, y propuso la semántica de ésta para los lenguajes de programación, pero no llevó a cabo una implementación. La primera implementación de este sistema fue realizada mediante la modificación del lenguaje *Lisp* [Steele90], denominándolo *3-Lisp* [Rivières84].

En la Figura 3.3 se muestra un ejemplo real de una torre de intérpretes. Se trata de la implementación de un intérprete de un lenguaje de alto nivel, en el lenguaje de programación *Java* [Gosling96].

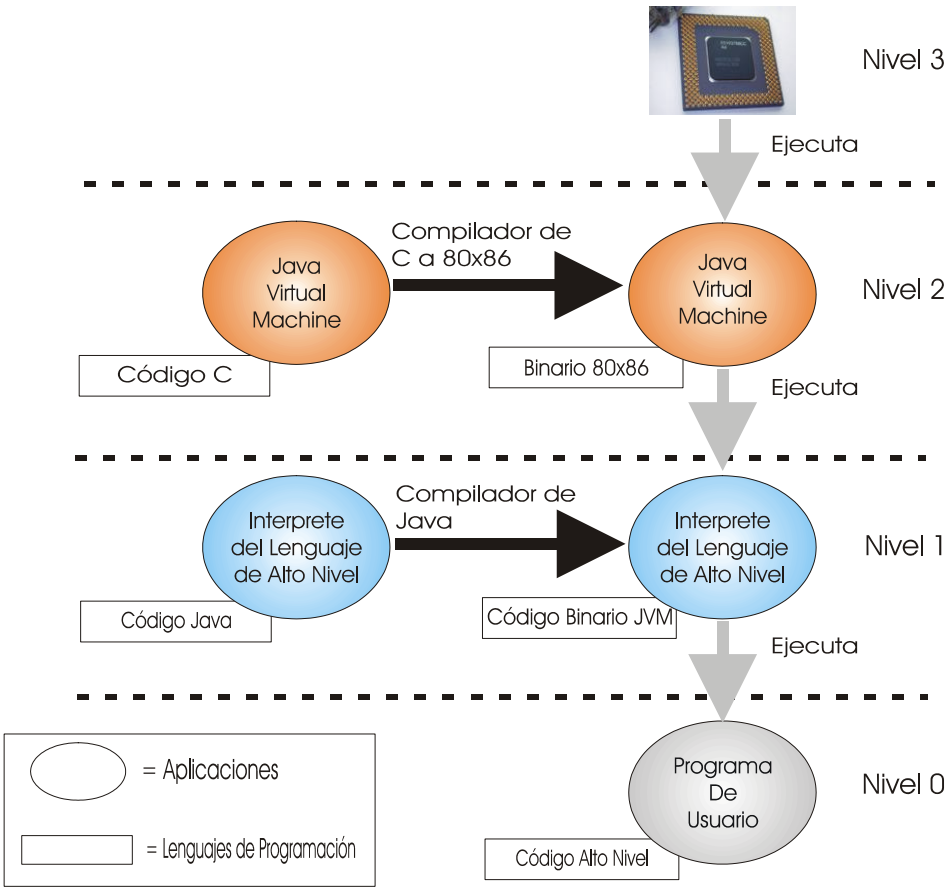

Figura 3.3: Torre de intérpretes en *Java*

Java es un lenguaje que se compila a código binario de una máquina abstracta denominada "*Java Virtual Machine*" [Lindholm96]. Para interpretar este código previamente compilado podemos utilizar un emulador de la máquina virtual programado en *C* y compilado, por ejemplo, para un *i80X86*. Finalmente el código binario de este microprocesador es interpretado una implementación física de éste. En este ejemplo tenemos un total de cuatro niveles computacionales en la torre de intérpretes.

Si pensamos en el comportamiento o la semántica de un programa como una función de nivel n, ésta podrá ser vista realmente como una estructura de datos desde el nivel n+1. En el ejemplo anterior, el intérprete del lenguaje de alto nivel define la semántica de dicho lenguaje, y la máquina virtual de *Java* define la semántica del intérprete del lenguaje de alto nivel. Por lo tanto, el intérprete de *Java* podrá modificar la semántica computacional del lenguaje de alto nivel, obteniendo así reflexión computacional.

Cada vez que en la torre de intérpretes nos movamos en un sentido de niveles ascendente, podemos identificar esta operación como cosificación[2] *(reificatio*n), mientras que un movimiento en el sentido contrario se identificará como reflexión *(reflectio*n) [Smith82].

2 *Se convierten comportamientos en datos (o cosas).*

3.3 CLASIFICACIONES DE REFLEXIÓN

Antes de exponer la clasificación propuesta, conviene destacar el hecho de que no es posible establecer una clasificación única de la reflexión, sino que pueden establecerse distintas clasificaciones del tipo de la misma que posee un sistema concreto en función de diversos criterios. En este punto se van a proponer estos criterios, se clasificarán los sistemas reflectivos en función de los mismos, y se describirá cada uno de ellos, siguiendo el trabajo desarrollado en [Ortin01].

3.3.1 Clasificación en Función de lo que se Refleja

Anteriormente hemos definido reflexión como la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo [Maes87]. Si tenemos en cuenta el grado de información que un sistema posee acerca de sí mismo, es decir, aquello que es susceptible de ser cosificado y reflejado, se puede establecer la siguiente clasificación:

3.3.1.1 INTROSPECCIÓN

La introspección es la capacidad de un sistema para poder inspeccionar u observar, pero no modificar, los objetos de un sistema [Foote92]. En este tipo de reflexión se facilita al programador acceder al estado del sistema en tiempo de ejecución, es decir, el sistema ofrece la capacidad de conocerse a sí mismo.

Esta característica se ha utilizado en numerosos sistemas muy extendidos para el desarrollo de todo tipo de aplicaciones. Por ejemplo, la plataforma *Java* [Kramer96] ofrece introspección mediante su paquete `java.lang.reflect` [Sun97]. Gracias a esta capacidad el lenguaje implementa ciertas operaciones que le dotan de funcionalidades muy útiles. Por ejemplo, en *Java* es posible almacenar un objeto en disco sin necesidad de implementar código adicional para ello: el sistema accede de forma introspectiva al objeto, analiza sus atributos y los convierte en una secuencia de *bytes* para su posterior envío a un flujo[3] [Eckel00]. Además, sobre este mismo mecanismo *Java* implementa su sistema de componentes *JavaBeans* [Sun96] con el que se puede, dado un objeto cualquiera en tiempo de ejecución, determinar su conjunto de propiedades y operaciones. Un sistema de similares características también está presente en el lenguaje *C#* [ECMA33405] de la plataforma *.NET* [Microsoft05] con el paquete `System.Reflection` [Microsoft06].

Otro ejemplo de introspección, en este caso para código nativo compilado, es la posibilidad de usar información en tiempo de ejecución incorporada al estándar *ISO/ANSI C++,* denominada *RTTI* (*Run-Time Type Information*) [Kalev98]. Este mecanismo introspectivo es mucho más limitado que los anteriores, ya que sólo permite conocer la clase de la que es instancia un objeto (en general, el tipo de un objeto dado), para poder ahormar éste de forma segura respecto al tipo [Cardelli97].

3 *Este proceso se conoce como "serialización" (serialization).*

3.3.1.2 REFLEXIÓN ESTRUCTURAL

En este nivel de reflexión también se refleja la estructura del sistema en tiempo de ejecución (las clases, el árbol de herencia, la estructura de los objetos y los tipos del lenguaje), pero se permite una mayor funcionalidad al soportar tanto su observación como su manipulación [Ferber88].

Mediante reflexión estructural se podrá acceder y conocer el estado de la ejecución de una aplicación desde el sistema base en un momento dado y modificar aquello que se estime oportuno. De esta manera, una vez reanudada la ejecución del sistema base después de producirse la reflexión, los resultados pueden ser distintos a los que se hubieren obtenido si la modificación de su estado no se hubiera llevado a cabo. Dichos cambios tendrán pues efecto en la computación a realizar por la aplicación, logrando así la conexión causal que se mencionaba anteriormente.

Ejemplos de sistemas de naturaleza estructuralmente reflectiva son *Smalltalk-80* [Goldberg83] y *Self* [Ungar87]. La programación sobre estas plataformas se centra en ir creando los objetos mediante sucesivas modificaciones de su estructura hasta formar el conjunto y composición deseada de los mismos. Una característica peculiar de este tipo de sistemas es que la "barrera" que separa tiempo de diseño y tiempo de ejecución es casi inexistente, ya que al poder acceder libremente a toda la estructura del sistema y modificar cualquier parte según sea necesario, manifestándose de inmediato el resultado de esos cambios, es muy difícil establecer dicha distinción. La figura 3.4 muestra un entorno de trabajo de *Squeak* (implementación de *Smalltalk)* donde se tiene acceso a toda la jerarquía de clases y estructura de la aplicación en tiempo de ejecución, modificando cualquier elemento de dicha estructura mediante el entorno gráfico soportado por este lenguaje.

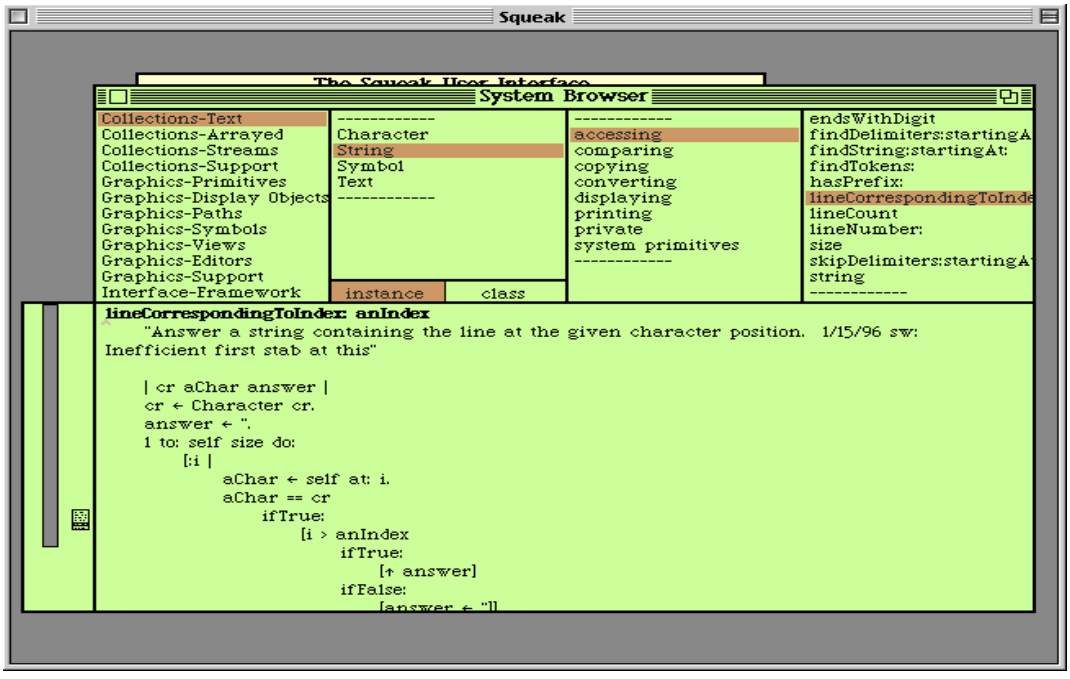

Figura 3.4: Captura de Pantalla del Entorno de Trabajo *Squeak* [Squeak05]

3.3.1.3 REFLEXIÓN COMPUTACIONAL

Este tercer nivel de reflexión es también denominado reflexión de comportamiento

*(behavioural reflectio*n). En este nivel se refleja el comportamiento exhibido por un sistema computacional, de forma que éste pueda computarse a sí mismo mediante un mecanismo de conexión causal [Maes87]. Un sistema dotado de reflexión computacional puede modificar su propia semántica, es decir, el propio comportamiento del sistema podrá cosificarse para su posterior manipulación.

Un ejemplo de abstracción de reflexión computacional es la adición de las *Proxy Classe*s [Sun99] a la plataforma *Java2*. Apoyándose en el paquete de introspección (`java.lang.reflect`), se ofrece un mecanismo para manipular dinámicamente la forma en la que un objeto interpreta la recepción de un mensaje, pudiendo de esta forma modificar la semántica del paso de mensajes de un objeto.

Otros ejemplos en la plataforma *Java* son la posibilidad de sustituir el modo en el que las clases se cargan en memoria y el grado de seguridad de ejecución de la máquina abstracta. Gracias a las clases `java.lang.ClassLoader` y `java.lang.SecurityManager` [Gosling96], se puede modificar la semántica de obtención del código fuente (de este modo se consigue cargar un *applet* de un servidor *web*) y el modo en el que una aplicación puede acceder a su sistema nativo (el sistema de protección frente a código malintencionado en la ejecución de *applet*s, conocido como *sandbox* [Orfali98]). Nótese que, aunque estos ejemplos son implementaciones de mecanismos que permiten reflexión computacional, el sistema que los implementa está limitando en todo momento qué elementos soportan este tipo de reflexión, es decir, no hay un soporte para este tipo de reflexión para todo el sistema, algo que nos encontraremos en la mayoría de los sistemas de este tipo que analizaremos en la segunda parte de este capítulo.

3.3.1.4 REFLEXIÓN LINGÜÍSTICA

Un lenguaje de programación cualquiera posee unas especificaciones léxicas [Cueva93], sintácticas [Cueva95] y semánticas [Cueva95b]. Un programa correcto es aquél que cumple las tres especificaciones descritas del lenguaje de programación con el que está implementado. Por otra parte, la semántica del lenguaje de programación identifica el significado de cualquier programa codificado en dicho lenguaje, es decir cuál será su comportamiento al ejecutarse. La reflexión computacional de un sistema nos permite cosificar y reflejar precisamente este último aspecto, su semántica, mientras que la reflexión lingüística [Ortín2001b] en cambio nos permite modificar cualquier aspecto del lenguaje de programación utilizado. Por tanto, en este nivel de reflexión desde el sistema base se podrá modificar el propio lenguaje de programación, lo que podría permitir, por ejemplo, añadir operadores, construcciones sintácticas o nuevas instrucciones al propio lenguaje desde el programa de usuario.

Un ejemplo de sistema dotado de reflexión lingüística es *OpenJava* [Chiba98]. Ampliando el lenguaje de acceso al sistema (*Java*) se añade una sintaxis y semántica para aplicar patrones de diseño [GOF94] de forma directa por el lenguaje de programación, amoldando el lenguaje a los patrones *Adapter* y *Visitor* [Tatsubori98].

3.3.2 Clasificación en Función de Cuándo se Produce el Reflejo

En la clasificación anterior es el sistema el que determina lo que puede ser cosificado para su posterior manipulación. En este caso se establecerá otra clasificación en función del momento en el que se puede llevar a cabo dicha cosificación.

3.3.2.1 REFLEXIÓN EN TIEMPO DE COMPILACIÓN

El acceso desde el sistema base al metasistema se realiza en el momento en el que el código fuente está siendo compilado, modificándose el proceso de compilación y por lo tanto las características del lenguaje procesado [Golm97].

OpenJava es una modificación de *Java* mencionada anteriormente y que le otorga capacidad de reflejarse en tiempo de compilación [Chiba98]. En este sistema se rectifica el compilador del lenguaje con una técnica de macros que expande las construcciones del mismo para, de esta forma, efectuar todas las operaciones reflectivas permitidas durante la compilación del programa. La eficiencia que se pierde por dotar de más flexibilidad al sistema recae pues sobre el tiempo de compilación, por lo que el coste de la introducción de más flexibilidad no penaliza la ejecución del programa como en otros sistemas, obteniendo por lo general un rendimiento superior en los programas realizados con sistemas que soportan este tipo de reflexión.

Por otra parte, aquello que se puede modificar en el sistema sigue estando dentro de alguno de los puntos de la clasificación anterior. En el ejemplo de *OpenJava* se pueden modificar todos los elementos: verbigracia, la invocación de métodos, el acceso a los atributos, operadores del lenguaje o sus tipos. No obstante, el hecho de que el acceso al metasistema se produzca en tiempo de compilación implica que una aplicación tiene que prever su flexibilidad antes de ser ejecutada, ya que una vez en ejecución no podrá accederse a su metasistema de una forma que no esté contemplada en su código fuente. El principal inconveniente es pues la limitación a la flexibilidad que se puede conseguir una vez la aplicación está en marcha, que obliga a que cualquier cambio necesario que no haya sido contemplado en tiempo de compilación requiera la modificación del código del programa y su posterior recompilación.

3.3.2.2 REFLEXIÓN EN TIEMPO DE EJECUCIÓN

En este tipo de sistemas el acceso al metasistema desde el sistema base, su manipulación y el reflejo producido por un mecanismo de conexión causal se llevan a cabo en tiempo de ejecución [Golm97]. En este caso, una aplicación tendrá la capacidad de ser flexible de forma dinámica, es decir, podrá adaptarse a cambios no previstos (modificación de requisitos, nuevas restricciones, reglas de negocio adicionales, etc.) cuando esté en plena ejecución.

Un ejemplo de este tipo de sistemas es *MetaXa*, que modifica la máquina virtual de la plataforma *Java* para poder ofrecer reflexión en tiempo de ejecución [Kleinöder96]. A las diferentes primitivas semánticas de la máquina virtual se pueden asociar metaobjetos que definan una nueva semántica a usar, anulando el funcionamiento anterior de dichas primitivas. Ejemplos clásicos de utilización de este tipo de flexibilidad son la modificación del paso de mensajes para implementar un sistema de depuración, auditoria o restricciones de sistemas en tiempo real [Golm97b].

3.3.3 Clasificación en Función de Cómo se Accede al Metasistema

En función del modo en el que se acceda al metasistema desde el sistema base, se puede establecer la siguiente clasificación:

3.3.3.1 REFLEXIÓN PROCEDURAL

La representación del metasistema se ofrece de forma directa por el programa que lo implementa [Maes87]. En este tipo de reflexión, el metasistema y el sistema base utilizan el mismo modelo computacional, de manera que si, por ejemplo, nos encontramos en un modelo de programación orientado a objetos, el acceso al metasistema se facilitará utilizando este paradigma. El hecho de acceder de forma directa a la implementación del sistema ofrece dos ventajas frente a la reflexión declarativa que veremos posteriormente:

- La totalidad del sistema es accesible y por lo tanto cualquier parte de éste podrá ser manipulada. No existen restricciones en el acceso.
- La conexión causal es automática [Blair97]. Al acceder directamente a la implementación del sistema base, no es necesario desarrollar un mecanismo de actualización o reflexión.

3.3.3.2 REFLEXIÓN DECLARATIVA

En la reflexión declarativa, la representación del sistema en su cosificación es ofrecida mediante un conjunto de sentencias representativas del comportamiento del mismo [Maes87], de manera que, haciendo uso de las mismas, el sistema podrá ser adaptado. La ventaja principal que ofrece esta clasificación es la posibilidad de elevar el nivel de abstracción a la hora de representar el metasistema. Las dos ventajas de la reflexión procedural comentadas previamente, se convierten en inconvenientes para este tipo de sistemas.

La mayor parte de los sistemas existentes ofrecen un mecanismo de reflexión declarativa. En el caso del sistema *MetaXa* [Kleinöder96] previamente mencionado, se ofrece una representación de acceso a las primitivas de la máquina abstracta modificables.

3.3.4 Clasificación en Función de Desde Dónde se Puede Modificar el Sistema

El acceso reflectivo a un sistema se puede llevar a cabo desde distintos procesos. La siguiente clasificación no es excluyente: un sistema podría estar incluido en ambas.

3.3.4.1 REFLEXIÓN CON ACCESO INTERNO

Los sistemas con acceso interno *(inward)* a su metasistema permiten modificar una aplicación desde la definición de éste (su codificación) [Foote92]. Esta característica define aquellos sistemas que permiten modificar una aplicación desde sí misma. La mayoría de los sistemas ofrecen esta posibilidad. En *MetaXa* [Kleinöder96], sólo la propia aplicación puede modificar su comportamiento.

3.3.4.2 REFLEXIÓN CON ACCESO EXTERNO

El acceso externo de un sistema *(outward*) permite la modificación de una aplicación mediante otro proceso distinto al que será modificado [Foote92]. En un sistema con esta característica, cualquier proceso puede modificar al resto. Su principal ventaja es la flexibilidad obtenida, pero presenta el inconveniente de que será necesario establecer un mecanismo de seguridad conveniente en el acceso entre procesos, ya que un mal uso de esta característica puede acarrear problemas de seguridad, inconsistencias y otras consecuencias graves para el comportamiento y estabilidad del sistema.

El sistema *Smalltalk-80* ofrece un mecanismo de reflexión estructural con acceso externo [Mevel87]: el programador puede acceder y modificar la estructura de cualquier aplicación o proceso del sistema.

3.3.5 Clasificación en Función de Cómo se Ejecuta el Sistema

La ejecución del sistema reflectivo puede darse de dos formas: mediante código nativo o mediante la interpretación de un código intermedio.

3.3.5.1 EJECUCIÓN INTERPRETADA

Si el sistema se ejecuta mediante una interpretación *software*, las características de reflexión dinámica se pueden ofrecer de una forma más sencilla. Ejemplos de este tipo de sistemas son *Self* [Smith95], *Smalltalk-80* [Krasner83] o *Java* [Kramer96]. Estas plataformas de interpretación ofrecen características reflectivas de un modo sencillo, al encontrase el intérprete y el programa ejecutado en el mismo espacio de direcciones. El principal inconveniente de este tipo de soluciones es la pérdida de eficiencia por la inclusión del proceso de interpretación, que será menos eficiente que un proceso basado en una compilación a código nativo y su posterior ejecución.

3.3.5.2 EJECUCIÓN NATIVA

La ejecución nativa se produce cuando se compila código nativo capaz de ofrecer cualquier grado de reflexión de los indicados. No debe confundirse esta clasificación con la establecida en función de cuando se produce el reflejo, ya que puede producirse el reflejo del sistema en tiempo de ejecución tanto para sistemas compilados (nativos) como interpretados. Un primer ejemplo de este tipo de sistemas es el mecanismo *RTTI* de *ANSI/ISO C++*, que ofrece introspección en tiempo de ejecución para un sistema nativo [Kalev98], aunque de una forma muy limitada. Otro posible ejemplo es *Iguana* [Gowing96], que implementa un compilador de *C++* que ofrece reflexión computacional en tiempo de ejecución. El código generado es una ampliación del estándar *RTTI*.

La principal ventaja de este tipo de sistemas es su mayor eficiencia. No obstante, su implementación es más costosa que la de un intérprete, puesto que debemos generar código capaz de ser modificado dinámicamente, labor que puede ser muy compleja. Muchos de estos sistemas limitan sus características reflectivas para ofrecer así una relación flexibilidad/coste de implementación razonable.

3.4 USO DE LA REFLEXIÓN: APLICACIONES

Una vez realizada una clasificación de los diferentes niveles y clases de reflexión, se pasará ahora a describir una panorámica de uso de la misma en diferentes sistemas actuales, basándose en los criterios establecidos en dicha clasificación. Este estudio ser realizará siguiendo el que podemos encontrar en [Ortin01]. En este apartado todos los sistemas a describir se estudiarán con la misma plantilla:

- Breve descripción del sistema concreto.
- Cómo se implementa la reflexión en el sistema.
- Beneficios y carencias generales (agrupados por cada apartado de la clasificación).

Además, se expondrá algún ejemplo práctico empleando los sistemas descritos en cada apartado de la clasificación, a modo de ejemplo ilustrativo de cada nivel de reflexión.

3.4.1 Sistemas Dotados de Introspección

Tal y como la hemos definido anteriormente, la introspección es la capacidad de acceder a la representación de un sistema, estando restringida esta capacidad a simplemente conocer este estado y no alterarlo, por lo que no cabe en este caso hablar de un mecanismo de conexión causal, ya que realmente no hay modificaciones posibles que se puedan reflejar para alterar el funcionamiento del sistema. Se puede decir pues que la introspección es una versión "simplificada" de la reflexión estructural, en la que sólo se permite leer el estado, pudiéndose denominar también "reflexión estructural de sólo lectura" [Foote90].

Al tener menos funcionalidad que el resto de tipos de reflexión, su implementación es más simple y hoy en día podemos ver ejemplos de la misma en lenguajes ampliamente extendidos como *C#* y *Java*, que describiremos en esta sección junto con otros. Además, mediante introspección se han desarrollado aplicaciones específicas que abarcan diversos campos, como la especificación de componentes, arquitecturas de objetos distribuidos, programación orientada a objetos o automatización de pruebas de *software*, por citar algunos ejemplos.

3.4.1.1 *ANSI/ISO C++ RUNTIME TYPE INFORMATION* (*RTTI*)

El lenguaje de programación *C++* es un lenguaje compilado que genera código nativo ejecutable sobre cualquier tipo de plataforma [Cueva98]. Tal y como se procesa y se genera dicho código, en la fase de ejecución del mismo no es posible acceder a información relativa al sistema ni conocer los tipos de los objetos en tiempo de ejecución. Para tratar de superar parcialmente esta carencia, el estándar *ANSI/ISO* de este lenguaje amplió su especificación, añadiendo cierta información del tipo (*RTTI*) en tiempo de ejecución, de forma que un programador pueda tener mayor seguridad respecto al tipo de un elemento concreto [Kalev98].

El caso típico en la utilización de este mecanismo es el ahormado o *cast*

descendente *(downcast)* en una jerarquía de herencia [Eckel00b]. Un puntero o referencia a una clase P (padre) se puede utilizar para apuntar a cualquier objeto de esa clase P o bien a uno de cualquiera de las clases derivadas de P (hijas) (Figura 3.5). No obstante, si tenemos un puntero de tipo P que apunta a un objeto de una clase derivada de P, sólo podremos pasarle aquéllos mensajes definidos en la clase padre P, ya que será la interfaz del objeto padre el único al que tendremos acceso mediante este puntero. Esto equivale a decir que se pierde el tipo del objeto instancia al que apunta, al no poder invocar a los métodos propios de la clase derivada.

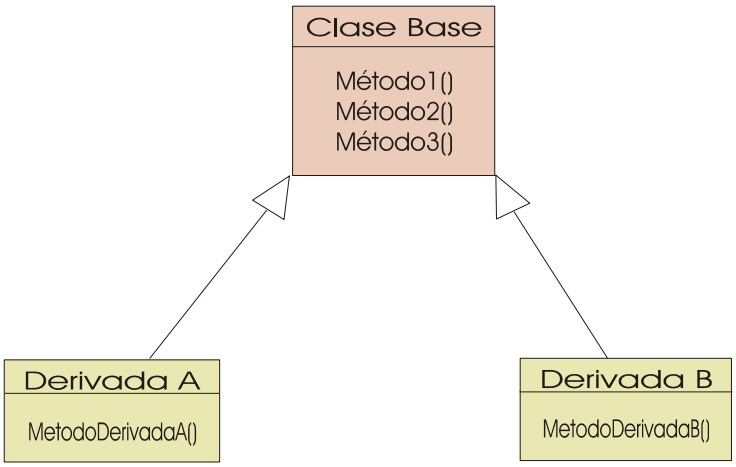

Figura 3.5: Punteros o referencias "Base" permiten la utilización genérica de objetos derivados

Una solución a este problema se lleva a cabo añadiendo a los objetos información dinámica relativa a su tipo (introspección). Mediante el estándar *RTTI* podemos preguntar a un objeto cuál es su tipo y recuperarlo si se hubiese perdido, como en el caso que se presentaba antes [Stroustrup98]. El siguiente fragmento de código, referente a la jerarquía de clases mostrada en la Figura 3.5, recupera el tipo del objeto gracias al operador `dynamic_cast` que forma parte del estándar *RTTI* mencionado.

```
Base *ptrBase;
DerivadaA *ptrDerivadaA=new DerivadaA;

// * Tratamiento genérico mediante herencia
ptrBase=ptrDerivadaA;
// * Solamente se pueden pasar mensajes de la clase Base
ptrBase->metodoBase();
// * Solicitamos información del tipo del objeto
ptrDerivadaA=dynamic_cast<DerivadaA*>(ptrBase);
// * ¿Es de tipo DerivadaA?
if (ptrDerivadaA)
    // * Recuperamos el tipo; se pueden pasar mensajes de Derivada
    ptrDerivadaA->mentodoDerivadaA();
```

Como vemos, al acceder al objeto derivado mediante el puntero base, se pierde el tipo de éste, no pudiendo invocar a los mensajes propios del objeto derivado. Gracias a la utilización de *RTTI*, el programador puede recuperar el tipo real del objeto e invocar a estos mensajes propios del mismo.

3.4.1.2 PLATAFORMA *JAVA*

La plataforma *Java* [Kramer96] define una interfaz de desarrollo de aplicaciones (*API*, *Application Programming Interface*) que dota de introspección al lenguaje. Esta parte del *API* se denomina *The Java Reflection API* [Sun97d] y permite acceder en tiempo de ejecución al estado de la máquina virtual, es decir, a la representación de las clases, interfaces y objetos existentes en la misma, lo que dota al sistema de las siguientes capacidades:

- Determinar la clase de la que un objeto es instancia.
- Obtener información sobre los modificadores de una clase [Gosling96] y su composición (métodos, atributos, constructores, clases base,...).
- Obtener una lista de métodos y constantes pertenecientes a cualquier *interface*.
- Crear una instancia de una clase totalmente desconocida en tiempo de compilación, lo que permite cargar dinámicamente clases que inicialmente no estaban cargadas al iniciar el programa.
- Obtener y asignar el valor de un atributo de un objeto en tiempo de ejecución, sin necesidad de conocer éste en tiempo de compilación.
- De la misma forma que con los atributos, se permite invocar un método de un objeto en tiempo de ejecución, aun siendo desconocido en tiempo de compilación.
- Crear dinámicamente un *array* y modificar los valores de sus elementos.

Sobre este *API* de introspección se define el paquete `java.beans`, que ofrece un conjunto de clases e *interfaces* para desarrollar la especificación de componentes *software* de la plataforma *Java*: los *JavaBeans* [Sun96]. La introspección en este caso permite al programador "leer" la estructura de cualquier componente en tiempo de ejecución, sin conocer de antemano los atributos y métodos del mismo. De esta forma, se puede desarrollar *software* que se adapte a cualquier componente en tiempo de ejecución y que sea capaz de proporcionar una determinada funcionalidad necesaria para una aplicación concreta, mediante un análisis que determine cómo se puede emplear.

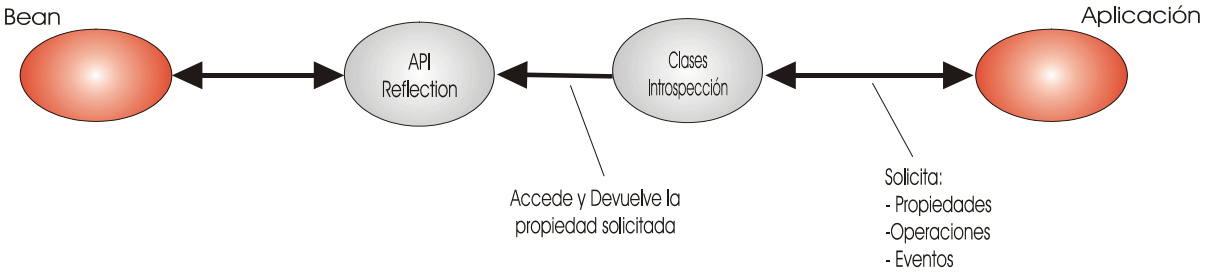

Figura 3.6: Utilización de introspección en el desarrollo de un sistema de componentes

El paquete de introspección es pues un medio seguro de acceder al *API* de reflexión para conocer y acceder a los métodos, propiedades y eventos de cualquier componente en tiempo de ejecución. En la Figura 3.6 se aprecia cómo una aplicación solicita el valor de una propiedad de un *Bean.* Mediante el paquete de introspección se usa la operación adecuada para ello y al invocarla devuelve el valor de la propiedad pedida para el objeto que se le proporciona.

Por último, se debe mencionar otra de las grandes aplicaciones de la reflexión dentro de la plataforma *Java*: poder convertir cualquier objeto en una secuencia de *bytes*[4] fácilmente transmisible vía red o almacenable en cualquier soporte físico o medio de almacenamiento (fichero, *BD*,...). Teóricamente, si un objeto en *Java* hereda del *interface* `java.io.Serializable` entonces podrá ser convertido a una secuencia de *bytes* usando el mecanismo de introspección para conocer el estado del objeto [Eckel00], sin necesidad de implementar ningún método adicional. El proceso seguido emplea el paquete `java.reflect` para consultar automáticamente el valor de todos los atributos del objeto y convertir éstos a *bytes*. Si un atributo de un objeto es a su vez otro objeto, se repite el proceso de forma recursiva. Siguiendo el proceso inverso es posible crear del mismo modo un objeto a raíz de un conjunto de *bytes*. Esto, que teóricamente es una forma sencilla de convertir cualquier objeto a una secuencia de *bytes*, ocasiona no obstante bastantes problemas y tiene múltiples peculiaridades que hacen que el proceso diste todavía de ser perfecto [Gene00], aunque es previsible que en un futuro cercano su implementación sea plenamente funcional.

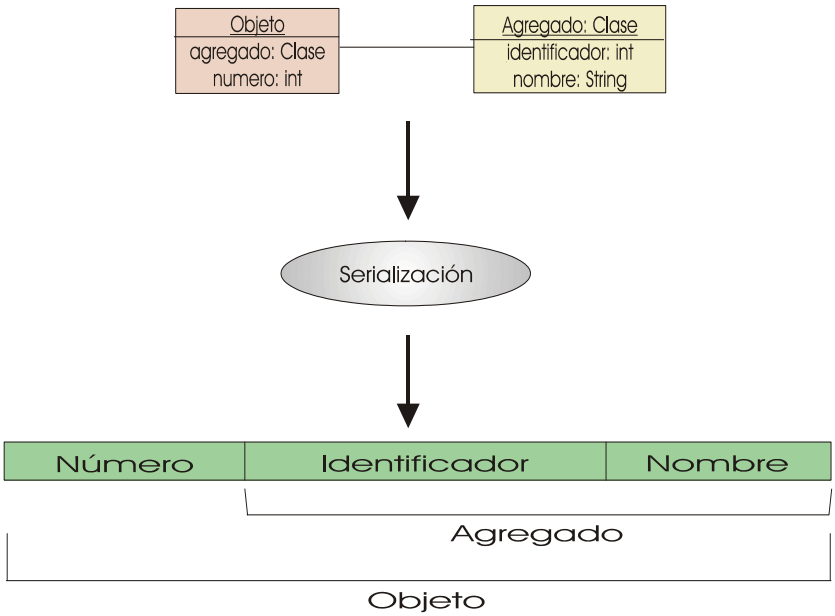

Figura 3.7: Conversión de un objeto a una secuencia de *bytes*, mediante un acceso introspectivo

3.4.1.3 PLATAFORMA *.NET*

La plataforma *.NET* [Microsoft05] es un entorno de desarrollo basado en el estándar *CLI* del *ECMA* [ECMA02], usando una máquina virtual que en su versión comercial se denomina *CLR* [Burton02]. Una descripción más detallada de esta plataforma se hará en el capítulo dedicado a máquinas virtuales, por lo que ahora sólo se hará mención a las capacidades introspectivas que ofrecen una serie de paquetes de la librería estándar que incorpora. En general puede afirmarse que las capacidades introspectivas proporcionadas por esta plataforma son muy similares en cuanto a funcionalidad y uso a las que ofrece *Java*, teniendo básicamente las mismas aplicaciones prácticas que se han visto anteriormente. No obstante, esta plataforma posee alguna

4 Proceso conocido como "serializacion" (serialization)

característica adicional, como es la programación generativa, que será descrita posteriormente, y que lo dota de mayor flexibilidad.

De forma muy similar a la plataforma *Java*, las clases del *API* estándar que proporcionan introspección en *.NET* permiten a cualquier lenguaje incorporado en la plataforma inspeccionar la estructura de cualquier clase del sistema, encontrar todos los tipos definidos en un ensamblado o *assembly* (entidad que es usada en esta plataforma para contener tipos) o bien obtener información acerca de cualquier miembro de cualquier clase. Las capacidades introspectivas de la plataforma se ubican dentro del espacio de nombres del *API System.Reflection***.** La clase *Type* identifica a una clase que ha sido reflejada, mientras que los miembros de la misma son identificados por la clase *MemberInfo***.** De igual modo, los métodos son reflejados a través de la clase *MethodInfo* y los parámetros de los mismos a través de la clase *ParameterInfo*. Por último, la clase *Activator* mediante el método *CreateInstance* permite cargar instancias en tiempo de ejecución.

A continuación se muestra un ejemplo en *C#* (uno de los lenguajes de la plataforma *.NET*) que permite averiguar todos los tipos contenidos en un *assembly*:

```
using System;
using System.Reflection;

class TiposAssembly
{
        public static void Main (string [] args)
        {
                // Obtener el assembly cuyo nombre se pasa como parámetro
                Assembly assembly = Assembly.LoadFrom (args[0]);

                // Obtener todos los tipos del assembly
                Type[] arrayTipos = assembly.GetTypes ();

                Console.WriteLine ("Tipos del ensamblado: " + args[0] + ":");
                Console.WriteLine ("");

                foreach (Type tipo in arrayTipos)
                {
                        // Namespace
                        Console.WriteLine ("Espacio de Nombres: " + tipo.Namespace);
                        // Nombre de la case
                        Console.WriteLine ("Clase: " + tipo.FullName );
                        // Clase padre
                        if (tipo.BaseType != null)
                                Console.WriteLine ("Clase padre: " + tipo.BaseType.FullName);
                        else Console.WriteLine ("Clase padre: <No tiene>");

                        Console.WriteLine ("|-----------------------------------------|");
                }
        }
}
```

Además, usaremos el lenguaje *C#* y la plataforma *.NET* para ilustrar un ejemplo práctico de uso de la introspección, empleando para ello la programación orientada a atributos o *Attribute Oriented Programming*, presente esta plataforma y también en la ultima versión de la plataforma *Java* (como una evolución de *Xdoclet* [XDoc2005]). Ésta se basa en el uso de atributos, que son etiquetas que permiten marcar elementos de un programa (clases, métodos,…) para indicar que mantienen una semántica adicional a nivel de aplicación o de dominio. De esta forma se permite separar la lógica de la aplicación (lógica de negocio) de otras lógicas, como servicios de log o bien designar métodos de servicios *Web*, que no lo son.

La programación orientada a atributos se basa en la incorporación de atributos a las diversas entidades del programa. Los atributos son metadatos, un conjunto de información declarativa asociada a las entidades *software* (clases, métodos, propiedades,...), pero que no introducen nueva semántica a dichas entidades. Los

mecanismos de introspección permiten precisamente leer estos atributos, por lo que se puede determinar exactamente cuando una entidad posee un atributo de un determinado tipo. Usando este mecanismo junto a la introspección podemos separar el código que representa la funcionalidad concreta de un programa de otras tareas adicionales ajenas, que podrían así ser reutilizadas en otros programas. A modo de ejemplo, en el siguiente ejemplo se verá cómo el código de un programa se separa de sus pruebas, pudiéndose reutilizar dicho código de prueba si fuese necesario:

```
public class Integer {//Clase que se va a probar
        private int theValue;
        public int Value {
                get { return theValue; }
                set { theValue=value; }
        }

        public Integer(int n) { theValue=n; }
        public Integer Add(Integer e) {
                return new Integer(theValue+e.Value);
        }
        public Integer Divide(Integer e) {
                return new Integer(theValue/e.Value);
        }
}

[TestFixture] public class IntegerTestNUnit {
        private Integer e1;
        private Integer e2;

        [SetUp] public void Init() {
                e1= new Integer(2);
                e2= new Integer(2);
        }

        [Test] public void Add() {
                Assert.AreEqual(e1.Value+e2.Value, e1.Add(e2).Value);
        }

        [Test] public void Divide() {
                Assert.AreEqual(e2.Value/e1.Value, e2.Divide(e1).Value);
        }

        [Test][ExpectedException(typeof(DivideByZeroException))] public void
        DivideByZero() {
                e1.Divide(new Integer(0));
        }
}
```

Este código ilustra un sencillo ejemplo de cómo se va a probar la clase `Integer`, separando la clase de su código de prueba, usando la herramienta comercial *NUnit V2* [NUnit05], que emplea conjuntamente para ello los atributos y la introspección. Vemos cómo se ha diseñado una clase de pruebas con una serie de métodos etiquetados con el atributo *Test*, que serán usados para probar los métodos correspondientes de dicha clase `Integer`, pudiendo además indicar otros datos, como por ejemplo el tipo de excepción que se espera que el método produzca. Además, el método etiquetado como *SetUp* será usado para inicializar aquellos datos necesarios para hacer dicho *test*. La propia clase de *test* está etiquetada con el atributo *TextFixture* para indicar precisamente que está dedicada a esta labor.

Siguiendo este convenio, *NUnit* o cualquier programa preparado para ello será capaz de acceder a esta clase y leer sus atributos y métodos mediante introspección, por lo que podrá fácilmente ejecutar las pruebas destinadas a la clase, haciendo todo el proceso de forma automática y presentando los resultados de dicho *test* al usuario sin que éste tenga que preocuparse de diseñar o lanzar cada uno de los *test*. Por otra parte, un programa como éste puede aceptar cualquier clase etiquetada siguiendo el convenio que se ha mostrado, siendo por tanto reutilizable su código para hacer pruebas de

cualquier clase, gracias a la introspección.

PROGRAMACIÓN GENERATIVA EN *.NET*

Además del ya descrito soporte para introspección, esta plataforma cuenta con una capacidad reflectiva adicional: La programación generativa. La programación generativa consiste en la generación dinámica de código fuente bajo demanda de forma automatizada, a través de herramientas como clases genéricas, plantillas, aspectos y generadores de código. Mediante su uso, cualquier programador podrá generar nuevos tipos e instancias de los mismos de forma dinámica a medida que el *software* las vaya necesitando, cargándolas en memoria y empleándolas como tipos "normales" definidos en el programa, con el objeto de aumentar su productividad. El conjunto de clases que permiten ese tipo de operaciones se encuentran en el paquete estándar `System.Reflection.Emit`. Éste es un ejemplo de generación dinámica de código en *C#* [CodeGuru05]:

```
using System;
using System.Reflection;
using System.Reflection.Emit;   // Para emitir CIL
using System.Threading;         // Para obtener el AppDomain actual

public class MSILGen
{
 public MSILGen()
 {
  // Creamos un nombre para el assembly.
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "CodeGenAssembly";

  // Create the dynamic assembly.
  AppDomain appDomain = Thread.GetDomain();
  AssemblyBuilder assembly = appDomain.DefineDynamicAssembly
                            (assemblyName,
                             AssemblyBuilderAccess.Run);

  // Creamos un módulo dinamicamente.
  ModuleBuilder module = assembly.DefineDynamicModule ("CodeGenModule");

  // Definimos una clase pública llamada "CodeGenClass" en el assembly.
  TypeBuilder helloWorldClass = module.DefineType("CodeGenClass", TypeAttributes.Public);

  // Definimos un atributo privado llamado "Message" en el tipo anterior.
  FieldBuilder greetingField = helloWorldClass.DefineField ("Message", typeof(String),
                               FieldAttributes.Private);

  // Creamos el constructor.
  Type[] constructorArgs = { typeof(String) };
  ConstructorBuilder constructor = helloWorldClass.DefineConstructor(
                                   MethodAttributes.Public,
                                   CallingConventions.Standard,
                                   constructorArgs);

  /* Generamos el CIL del metodo. El constructor llamará al constructor de su clase padre y
guarda su argumento en el atributo privado.*/
  ILGenerator constructorIL = constructor.GetILGenerator();
  constructorIL.Emit(OpCodes.Ldarg_0);

  ConstructorInfo superConstructor = typeof(Object).
                  GetConstructor(new Type[0]);

  constructorIL.Emit(OpCodes.Call, superConstructor);
  constructorIL.Emit(OpCodes.Ldarg_0);
  constructorIL.Emit(OpCodes.Ldarg_1);
  constructorIL.Emit(OpCodes.Stfld, greetingField);
  constructorIL.Emit(OpCodes.Ret);

  // Creamos el método GetMessage.
  MethodBuilder getGreetingMethod = helloWorldClass.DefineMethod("GetMessage",
```

```
                                    MethodAttributes.Public,
                                    typeof(String), null);

  // Generamos el CIL para el método GetGreeting.
  ILGenerator methodIL = getGreetingMethod.GetILGenerator();
  methodIL.Emit(OpCodes.Ldarg_0);
  methodIL.Emit(OpCodes.Ldfld, greetingField);
  methodIL.Emit(OpCodes.Ret);

  // Creamos la clase CodeGenClass.
  typ = helloWorldClass.CreateType();
 }

 Type typ;

 public Type T
 {
   get { return this.typ; }
 }
}

using System;
using System.Reflection;
using System.Reflection.Emit;

class Class1
{
  static void Main()
  {
    MSILGen codeGen = new MSILGen();

    Type typ = codeGen.T;

    // Creamos una instancia de la clase "CodeGenClass".
    object codegen = Activator.CreateInstance(typ,
                                        new object[] {"Hola desde CodeGenClass."});

    // Invocamos el metodo "GetMessage" de la clase "CodeGenClass".
    object obj = typ.InvokeMember("GetMessage", BindingFlags.InvokeMethod, null,
                                  codegen, null);

    Console.WriteLine("CodeGenClass.GetMessage retorna: \"" + obj + "\"");
  }
}
```

Esta capacidad dota a los lenguajes de la plataforma *.NET* de nuevas funcionalidades, ya que permitiría a un usuario involucrarse en la creación de código, creando nuevas clases y reglas de negocio, o bien generar herramientas que hagan este trabajo. No obstante, no puede afirmarse que la plataforma *.NET* ofrezca capacidades de reflexión estructural completas empleando este mecanismo, ya que posee ciertas limitaciones que impiden clasificarlo como tal:

- Sólo permite crear tipos nuevos, aunque pueden derivarse de otros ya existentes.
- No se permite la modificación de tipos existentes, es decir, no es posible añadir nuevos atributos y métodos (ni por supuesto modificarlos o eliminarlos) a tipos ya cargados en memoria. Es más, una vez que un tipo nuevo se crea y se habilita para ser usado como un tipo cualquiera, es imposible alterar la composición de ningún miembro.
- El motivo de la limitación anterior es que los tipos nuevos poseen dos "estados": En construcción (momento en el que se pueden alterar los miembros de su estructura) y construidos (momento en el que se manejan como tipos estándar y ya no se pueden modificar). No hay pues una verdadera noción de tipo dinámico.
- No se ofrece conocimiento de código existente.

Por tanto, este mecanismo es limitado y de aplicación restringida, no siendo equiparable con un verdadero soporte de reflexión estructural completo, aunque está por encima de la introspección.

3.4.1.4 DYLAN

Dylan es un lenguaje de programación orientado a objetos, funcional, dinámico que soporta el desarrollo rápido de programas. Si es necesario los programas desarrollados con el mismo pueden ser optimizados para una ejecución más eficiente suministrando información de tipos adicional al compilador. Casi todas las entidades en *Dylan* (funciones, clases, tipos básicos de datos,…) son objetos de primera clase. También soporta herencia múltiple, polimorfismo, introspección de objetos, macros y otras características avanzadas [Gwydion05]. Describiremos a continuación una breve reseña acerca del modelo de clases empleado por este lenguaje y sus principales características, para luego describir qué tipo de funcionalidades reflectivas podemos encontrar en el mismo.

Dylan es un lenguaje que intenta integrar las mejores características de la programación orientada a objetos y procedural así como de los lenguajes dinámicos y estáticos intentando evitar sus inconvenientes [Feinberg97]. En su diseño se han tenido en cuenta los siguientes objetivos:

- Facilitar la programación modular con programas formados por componentes reusables.
- Proporcionar soporte para la programación por procedimientos.
- Facilitar el desarrollo rápido de programas, que a su vez sean rápidos y compactos.

ATRIBUTOS Y CLASES

Tanto *Dylan* como *CLOS* tienen la característica de que en su modelo de objetos las clases pueden heredar de varias clases padre (herencia múltiple) y por tanto deben emplear un mecanismo para seleccionar qué método se va a ejecutar en un momento dado si se plantean diferentes alternativas. En *Dylan* las clases pueden incluir *slots*, que representan atributos o miembros de datos de forma similar a lo que implementan otros lenguajes orientados a objetos. Todos los accesos a *slots* se hacen usando métodos, proporcionándose métodos *get* y *set* (para obtener y cambiar el valor) por defecto para cada uno de ellos. Además *Dylan* incorpora una solución intermedia entre los sistemas de tipos estáticos y dinámicos para la declaración de atributos y variables, al permitir omitir la declaración del tipo de una variable o atributo si así se desea, pero no prohibiéndola, de forma que un programador podrá decidir si sacrifica flexibilidad por una comprobación de tipos en tiempo de compilación que ofrezca un mejor rendimiento en la ejecución de un programa. Este ejemplo muestra cómo se declaran los atributos en *Dylan* [Gwydion05]:

```
define class <window> (<view>)
   slot title :: <string> = "untitled", init-keyword: title:;
   slot position :: <point>, required-init-keyword: position:;
 end class;
```

En él se puede ver como la clase *window*, derivada de la clase *view*, define dos atributos o *slots* declarando su tipo y un valor inicial en algunos casos, así como también unas *keywords* que serán usadas en la creación de un objeto si se pasan como argumento en dicha creación, para dar valor inicial a esos atributos en ese caso. Nótese también cómo se puede hacer obligatorio el uso de *keywords* para ciertos atributos si así lo desea el programador de la clase.

FUNCIONES GENÉRICAS Y MÉTODOS: MECANISMOS DE INTROSPECCIÓN

El modelo de objetos de *Dylan* contempla además dos conceptos para trabajar con mensajes: Las funciones genéricas y los métodos. Una función genérica representa un conjunto de métodos similares. Cada método creado mediante `define method` automáticamente se hace pertenecer a una función genérica del mismo nombre. Por ejemplo, un usuario puede definir tres métodos `mostrar(…)`, cada uno de ellos con un parámetro de un tipo diferente, y todos ellos formarán parte de una misma función genérica que tendrá ese mismo nombre. Este código ilustra el ejemplo [Gwydion05]:

```
define method mostrar(i :: <integer>)
  mostrar-integer(i);
end;

define method mostrar(s :: <string>)
  mostrar-string(s);
end;

define method mostrar(f :: <float>)
  mostrar-float(f);
end;
```

Cuando un programa llama a `mostrar(…)` *Dylan* examina los métodos que tiene disponibles y selecciona el adecuado en función de los parámetros proporcionados, pero todo ello se realiza en tiempo de ejecución, a diferencia de otros lenguajes como *C++*. Si no hay ningún método adecuado, se generaría un error. Este concepto es conocido como multimétodo.

Dylan incorpora una serie de funcionalidades de introspección apoyadas sobre funciones genéricas como las vistas, que permiten examinar objetos, clases y funciones [Gwydion05b] [Gwydion05c]:

- Es posible determinar la clase de un objeto llamando a `object-class`.
- Si se desea averiguar si un objeto es instancia de una clase X determinada, se empleará `instance?`.
- También es posible obtener una secuencia de clases de las que hereda una clase dada directamente (ya hemos mencionado que se permite la herencia múltiple), empleando `direct-superclasses`, mientras que si es necesario obtener todos los ancestros de una clase dada se puede emplear `all-superclasses`.
- Para la localización de un método, *Dylan* implementa la función `find-method`. Cuando se pasa el nombre de una función genérica y una lista de clases a este método, se devuelve el método asociado con la función que corresponde a la lista de clases pasada. Por ejemplo, en el siguiente ejemplo se devuelve un método que añade dos instancias del tipo `<small-integer>`:

```
(find-method binary+ (list <small-integer> <small-integer>))
```

Por otra parte, si no existiese un método adecuado para los datos que se le han pasado a `find-method`, devolvería el valor booleano `false`, permitiendo al programador responder por código a este tipo de incidencias sin necesidad de abortar la ejecución. Además, las funciones en *Dylan* son objetos de primera clase (*first class objects*), lo que abre la puerta a la implementación de funciones reflectivas adicionales.

- La función `generic-function-methods` devuelve una secuencia de todos los métodos definidos para una función genérica.
- `method-specializers` permite obtener una secuencia de *specializers* para un método concreto, siendo un s*pecializer* de un método aquel objeto usado para seleccionar un método concreto cuando su función genérica es llamada. En la mayoría de los casos esta secuencia estará compuesta por clases, pero en *Dylan* es posible especializar un método para una instancia individual, por lo que en algunos casos las secuencias pueden devolver también objetos que no son clases.
- `function-arguments` que devuelve tres valores: el número de argumentos requerido por una función concreta, un booleano que indica si la función acepta argumentos adicionales y una secuencia que especifica palabras clave que la función puede emplear como parámetro (posiblemente vacía). En caso de no aceptar parámetros que sean palabras clave, el valor booleano `false` se pasa en vez de esta secuencia.
- `applicable-method?` Devuelve `true` si existe un método definido en una función que corresponda a un conjunto concreto de argumentos que se especifican y `sorted-applicable-methods` devuelve todos los métodos que pueden ser aplicados a un conjunto de argumentos.

Vemos pues cómo este lenguaje posee un conjunto de funciones que le capacita para tener un soporte completo de introspección adaptado a las características y particularidades del propio lenguaje y de su sistema de tipos, pero equivalente al poseído por otros sistemas que gozan del mismo nivel de reflexión.

EVENTOS Y CAPACIDADES REFLECTIVAS

Dylan posee además un modelo de eventos que se aprovecha de sus capacidades reflectivas. El sistema de eventos se implementó basándose en la orientación a objetos, donde existe una cola llena de objetos de varias clases evento. Los eventos son manejados eliminándolos de la cola y pasándoselos a un manejador genérico de eventos que los procesa. Además, se define una cola de manejadores que poseen funciones que manejan eventos. Cuando el manejador genérico obtiene un evento, puede extraer la información de los parámetros del evento y buscar en la cola de manejadores de eventos una función que se aplica a dichos parámetros. Para determinar si el manejador puede aplicarse o no, se usan las funciones antes vistas `applicable-method?` o `sorted-applicable-methods`.

Ésta es una versión muy simplificada del bucle que especifica el manejo de eventos de *Dylan*. Primero se define la función que encuentra el manejador apropiado:

```
(define-method find-event-handler (type    time
                                   location
                                   message)
    (any? (method (handler)
          (applicable-method?   (object-class type)
```

```
                        (object-class time)
                        (object-class location)
                        (object-class message)))
        *event-handler-queue*))
```

La función *any?* devuelve el primer elemento de la secuencia **event-handler-queue** para el que la función *applicable-method?* devuelve *true*. Por tanto, esta función busca en la cola de manejadores de eventos devolviendo el primer manejador apropiado a los parámetros que se le pasan. El bucle de manejador de eventos es:

```
(define-method event-loop ()
(for ((continue #t (test-whether-to-continue)))
        ((not continue) 'done)
    (bind ((current-event (pop-event-queue))
                (event-type event-time
                 event-location event-message
                    (event-params current-event))
                (handler (find-event-handler    event-type
                                                event-time
                                                event-location
                                                event-message)))
            (if handler
                (apply handler current-event)
                (apply *default-event-handler* current-event)))))
```

Este bucle itera hasta que algo modifica la variable *continue* y la pone a *false*. *bind* crea la variable *current-event* para contener el primer elemento de la cola de eventos, y entonces vincula las variables locales a los parámetros devueltos por la función *event-params*. Después se vincula el manejador al resultado de la función *find-event-handler* anteriormente definida. Finalmente, si *find-event-handler* ha encontrado un manejador adecuado entonces aplica dicho manejador sobre el evento actual o bien aplica el manejador por defecto definido por el sistema.

Como se puede apreciar en el código mostrado, el sistema de eventos de *Dylan* usa extensiblemente las capacidades reflectivas del lenguaje. Gracias a esto, el sistema de eventos descrito puede ser muy flexible y extensible, ya que los programadores pueden disponer de un grado de libertad elevado para definir nuevos tipos de eventos y manejadores propios en sus aplicaciones.

El modelo de objetos de *Dylan* muy influenciado por el diseño de *CLOS*. *CLOS* va más lejos en los servicios reflectivos que proporciona, al permitir, por ejemplo, redefinir la estructura de las clases. De esta forma, es posible cambiar la clase de una instancia usando *change-class*, o bien reejecutar el código que inicializa un objeto mediante *reinitialize-instance*, pero estas capacidades superan el nivel de reflexión visto en esta sección.

3.4.1.5 Boo

Boo [Boo05] es un lenguaje orientado a objetos basado en clases con un sistema de tipos que combina tipos estáticos y dinámicos. Además, está diseñado para el estándar *CLI* y altamente inspirado en *Python*. Existe un alto grado de similitud entre ambos lenguajes, pero mencionaremos aquí sólo las diferencias relativas al soporte de reflexión ofrecido y otras características importantes.

Una de las mayores diferencias entre *Boo* y *Python* es que *Boo* posee un sistema de tipos que, aunque no obliga al programador a declarar el tipo de todas y cada una de las variables que se usen en el programa (ya que cuenta con un mecanismo de inferencia

automática de tipos) intenta obtener las ventajas tanto de los tipos estáticos como de los dinámicos (por ejemplo, no se pierde ni seguridad ni rendimiento al declarar tipos). El mecanismo de inferencia de tipos está implementado de forma que se pone en marcha en la propia declaración de variables o atributos, *arrays*, métodos, tipos de retorno de métodos, etc., infiriendo el tipo de las variables declaradas en función de los valores que se le asignen.

Además de los tipos "estándar" como *object*, *int*, etc. *Boo* posee también un tipo especial llamado *duck*. Si se declara un objeto como de tipo *duck*, o bien es convertido mediante *cast* a ese tipo, el lenguaje no tratará de comprobar la existencia de cualquier operación que se llame sobre ese objeto en tiempo de compilación. En lugar de eso, convertirá la operación en métodos que no serán resueltos hasta tiempo de ejecución. Por tanto, el lenguaje realmente confía en tiempo de compilación en que la llamada es correcta (es decir, que un tipo *duck* puede realmente ser cualquier cosa) y posteriormente, con el programa en funcionamiento, hará las comprobaciones pertinentes. A este concepto se le denomina *duck typing*.

El siguiente ejemplo [Boo05] muestra el uso de tipos que hace este lenguaje:

```
static1 as int  //El tipo de la variable es entero
dynamic1 as duck //Puede ser cualquier cosa

static1 = 0
dynamic1 = 0
print static1+1 //-> 1
print dynamic1+1 //-> 1

#static1 = "Un string" //error, no se puede convertir de string a int
dynamic1 = "Un string" //dynamic1 puede ser cualquier tipo

#print static1.ToUpper() //error, a1 es un int, no un string
print dynamic1.ToUpper() //-> "Un string"

//Se puede convertir un tipo a duck:
dynamic2 as duck = static1
print dynamic2 //-> 0
dynamic2 = "Un string"
print dynamic2.ToUpper()//-> "Un string"

//Tambien se puede conviertir un tipo duck a otro tipo concreto.
static2 as string = dynamic1
print static2.ToUpper()

#static3 as int = dynamic1 //error, no se puede convertir un string a int
#print static3 + 2
```

Este lenguaje también incorpora un concepto que será explicado aquí y que está presente en varios lenguajes que también analizaremos en esta tesis: La cláusula o ***closure*** [Codehaus06]. Éstos son trozos de código ejecutables que pueden tener un estado asociado al mismo. Una de sus características principales es que pueden ser reutilizados en el programa, es decir, permiten parametrizar código de la aplicación. Estos elementos están formados por la combinación del código de una función con un ámbito especial (un conjunto de variables) vinculado a esa función. Las variables de un *closure* están vinculadas a la invocación de ese código, que puede acceder a ellas libremente en cada llamada. Este lenguaje soporta cláusulas mediante objetos `Proc`.

Además de otras diferencias sintácticas y de empleo de ciertos elementos del lenguaje [Boo05b], en lo que concierne a reflexión podemos mencionar que *Boo* sólo posee introspección, siendo ésta una de las grandes diferencias con el lenguaje *Python*. Una de las principales ventajas de *Boo* es que, al estar integrado en el *CLI,* puede usar e interoperar con código existente creado con cualquier lenguaje que esté diseñado para el mismo, permitiéndose además usar otros elementos del *CLI* como *locks*, atributos, etc.

Por tanto, el diseño de este lenguaje hace que no posea ciertas características de

Python (menor nivel de flexibilidad), pero esto se hace en aras a lograr un rendimiento superior a éste.

3.4.1.6 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Contar con características introspectivas dentro de una plataforma permite la implementación de funcionalidades que aportan importantes ventajas. Desde el punto de vista del programador, proporciona diversas funcionalidades muy útiles como hacer operaciones seguras respecto al tipo en tiempo de ejecución (*RTTI* de *C++*) o almacenar objetos en disco ("Serialización" de la plataforma *Java*). Desde el punto de vista de un sistema operativo, son necesarias para desarrollar un sistema de componentes (*.NET*), mientras que desde el punto de vista de entornos distribuidos permiten desarrollar aplicaciones flexibles (*Java*).

La utilización de la introspección en aplicaciones reales queda patente con numerosos ejemplos prácticos en sistemas actuales, algunos de ellos presentados en esta sección, como el ejemplo de *test* automatizado. En el desarrollo de aplicaciones de comercio electrónico, es común utilizar la transferencia de información independiente de la plataforma y autodescriptiva usando para ello el bien conocido y muy usado actualmente formato *XML* [W3C98]. Mediante la utilización de una librería (*DOM* [W3DOM06] o *SAX* [Megginson00]) es posible conocer los datos en tiempo de ejecución (introspección), eliminando el acoplamiento que se produce entre el cliente y el servidor en muchos sistemas distribuidos, por ejemplo. De esta forma, las aplicaciones ganan en flexibilidad pudiéndose adaptar a los futuros cambios.

De todas formas, no todo son ventajas. El carácter de "sólo lectura" de la introspección hace que la flexibilidad que se pueda alcanzar empleando este mecanismo sea limitada, de manera que la aplicación de la misma no sirva para determinados propósitos, por no poder alterar el estado del sistema y reflejar dichos cambios para lograr acciones más avanzadas mediante el empleo de este grado de flexibilidad. También es reseñable que este tipo de operaciones causan una penalización de rendimiento, por el coste adicional que tiene extraer de la propia información del programa aquellos atributos y métodos que son solicitados y hacer uso de ellos a través de los mecanismos de introspección. En este último caso será necesario hacer una búsqueda entre las estructuras de datos que contienen la información de las clases, y éste es un mecanismo menos eficiente que el acceso directo a los mismos en el código del programa.

3.4.2 Sistemas Dotados de Reflexión Estructural

Según lo que se ha definido anteriormente, un sistema dotado de reflexión estructural es aquél que posee la capacidad de acceder y modificar su estructura, de manera que esas modificaciones posteriormente produzcan cambios que se vean reflejados en la ejecución del programa. Estudiaremos a continuación la aplicación de este nivel de reflexión en diversos sistemas existentes.

3.4.2.1 SMALLTALK-80

Smalltalk es uno de los primeros entornos de programación que incorporó características de reflexión estructural. El sistema *Smalltalk-80* se puede dividir en dos elementos principales que trabajan conjuntamente:

- La imagen virtual, que es una colección de objetos que proporcionan funcionalidades de diversos tipos.
- La máquina virtual, que interpreta la imagen virtual y las aplicaciones de usuario.

En un principio la imagen virtual es cargada en memoria y su código es interpretado por la máquina. Si deseamos tener acceso a algún dato de cualquiera de las clases existentes, como su estructura, su descripción, el conjunto de los métodos que posee o incluso la implementación de éstos, podemos utilizar una aplicación desarrollada e integrada en el sistema denominada *browser* [Mevel87] (Figura 3.8). En este lenguaje todas las clases se tratan como objetos en tiempo de ejecución y son instancias de una clase X, que a su vez deriva de la clase predefinida `Class`. El *browser* es una aplicación integrada en *Smalltalk* que puede acceder a todos los objetos clase existentes en el sistema para inspeccionar y modificar su estructura, descripción y métodos asociados. En la figura 3.8 se muestra un caso particular de uso del *browser* integrado en *Smalltalk*, empleándolo para mostrar la documentación de un método cualquiera dinámicamente.

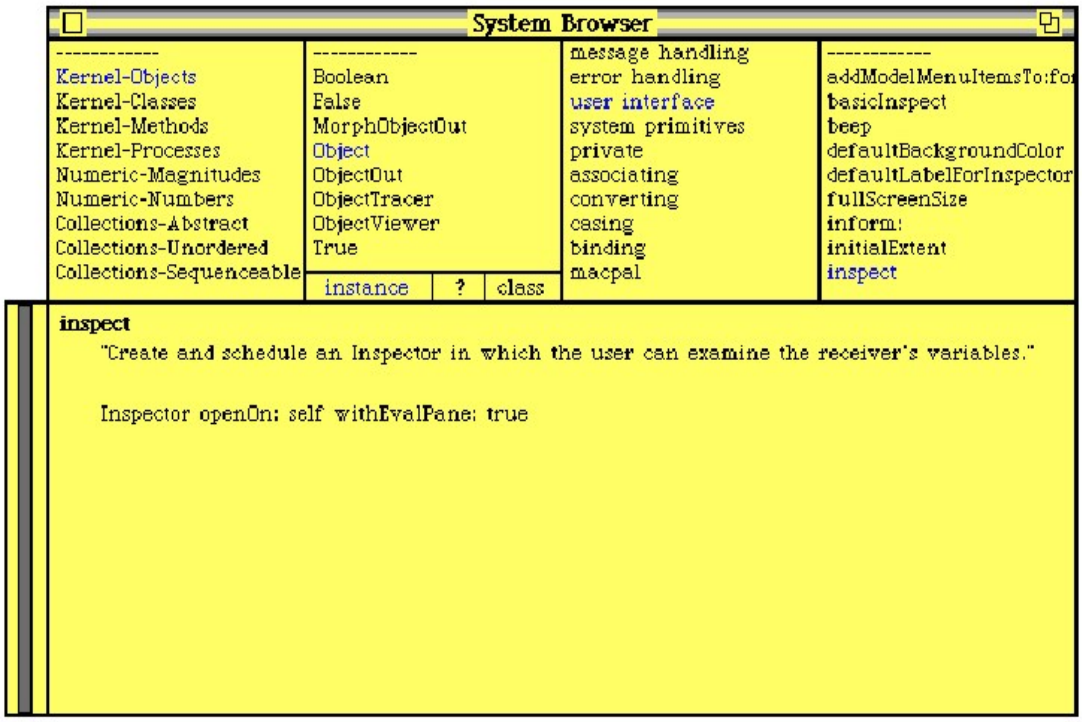

Figura 3.8: Análisis del método `inspect` del objeto de clase `Object`, con la aplicación `Browser` de *Smalltalk-80* [Ortin01]

Todas las operaciones que pueden realizarse sobre clases pueden también realizarse sobre los objetos existentes en tiempo de ejecución, usando el mensaje `inspect` para acceder al contenido de cada elemento [Goldberg89]. El *browser* es pues la

aplicación que permitirá hacer todas las operaciones necesarias con este lenguaje, ya sean reflectivas o no (por ejemplo, modificar el valor de un atributo de un objeto).

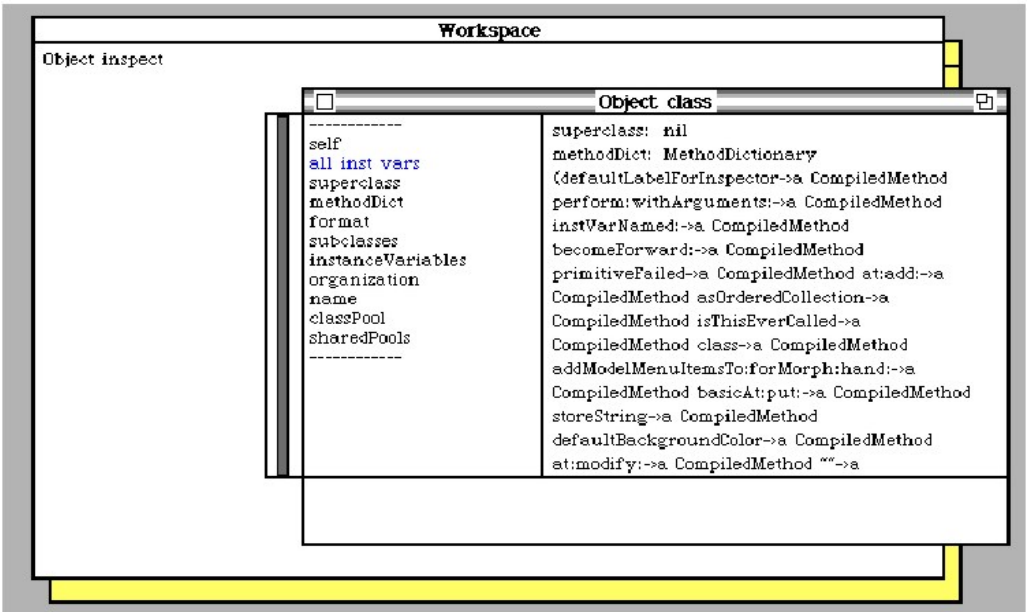

Figura 3.9: Invocando al método *`inspect`* del objeto *`Object`* desde un espacio de trabajo de *Smalltalk-80* [Ortin01]

En la Figura 3.9 se muestra cómo es posible hacer uso de la reflexión estructural para modificar la estructura de una aplicación en tiempo de ejecución. Por ejemplo, se podría utilizar el mensaje *`inspect`* para su depuración, permitiendo el acceso a las distintas partes de una instancia cualquiera sin necesidad de generar código adicional a la hora de compilar.

La finalidad del diseño de *Smalltalk* era obtener un sistema fácilmente manejable por personas que no fuesen informáticos, empleando para ello el paradigma de la orientación a objetos y la reflexión estructural. Una vez diseñada una aplicación consultando la información necesaria de las clases en el *browse*r, se puede depurar ésta accediendo y modificando los estados de los objetos en tiempo de ejecución. Si ocurriese algún error en el código del programa, simplemente se accedería al método de la clase donde esté localizado y se modificaría, reflejándose los cambios en el programa automáticamente haciendo uso de la reflexión estructural en tiempo de ejecución.

Por tanto, puede afirmarse que en *Smalltalk-80* se utilizaron los conceptos de reflexión estructural e introspección para obtener un entorno sencillo de manejar y autodocumentado. No obstante, dado que la semántica del lenguaje *Smalltalk* viene dada por una serie de primitivas básicas de computación de la máquina virtual, y éstas no son modificables por el programador, este lenguaje carece de reflexión computacional.

3.4.2.2 *SELF*, PROYECTO *MERLIN*

El sistema *Self* fue construido como una simplificación del sistema *Smalltalk,* y está también constituido por una máquina virtual y un entorno de programación basado en el lenguaje *Self* [Ungar87]. La reducción principal respecto a *Smalltalk* se centró en la eliminación del concepto de clase, dejando sólo la abstracción del objeto, incorporándose el concepto de "lenguaje basado en prototipos" que describiremos ampliamente en el

capítulo siguiente de esta tesis. Este sistema, orientado a objetos puro e interpretado, fue utilizado para estudiar técnicas de optimización de lenguajes orientados a objetos que se usan hoy en día en lenguajes comerciales [Chambers91]. En él se probaron métodos de compilación continua y compilación adaptable [Hölzle94] que hoy en día se usan en plataformas comerciales como *Java* [Sun98] y que describiremos en un capítulo posterior. Los últimos desarrollos derivados de este lenguaje tratan de extender el lenguaje a más plataformas nativas entre otras mejoras [Spitz06], y también existen desarrollos que construyen una máquina virtual para *Self* con el propio lenguaje *Self*, como parte del proyecto *Klein* [SunKlein06].El proyecto *Merlin*, creado con la pretensión de que los ordenadores fuesen utilizados fácilmente por los humanos, ofreciendo características como sistemas de persistencia y distribución implícitos [Assumpcao93], fue desarrollado en lenguaje *Self*.

Para conseguir un elevado grado de flexibilidad en el lenguaje *Self*, se trató de ampliar la máquina virtual del mismo con un conjunto de primitivas de reflexión *(region*s, *reflector*s, *metaspace*s) [Cointe92]. No obstante, la complejidad de ésta ampliación creció de tal forma que su portabilidad a distintas plataformas se convirtió en una tarea muy compleja [Assumpcao95]. Para tratar de solventar esta carencia, Assumpçao propone en [Assumpcao95] el siguiente conjunto de pasos para construir un sistema portable con capacidades reflectivas:

- Implementar, sobre cualquier lenguaje de programación, un pequeño intérprete de un subconjunto del lenguaje a interpretar. En este caso se creó un intérprete de "*tiny-Self*" que poseía la característica de reflexión estructural.
- Sobre lenguaje (*tinySelf*), se desarrolla un intérprete del lenguaje buscado (*Self*), con todas sus características.
- Implementamos una interfaz de modificación de las operaciones deseadas. La codificación de un intérprete en su propio lenguaje tiene la ventaja de ofrecer una flexibilidad total: Todo su comportamiento se podría modificar, incluso las primitivas computacionales, puesto que *tinySelf* posee reflexión estructural dinámica. Sólo debemos implementar un protocolo de acceso adecuado a las operaciones que deseemos modificar.

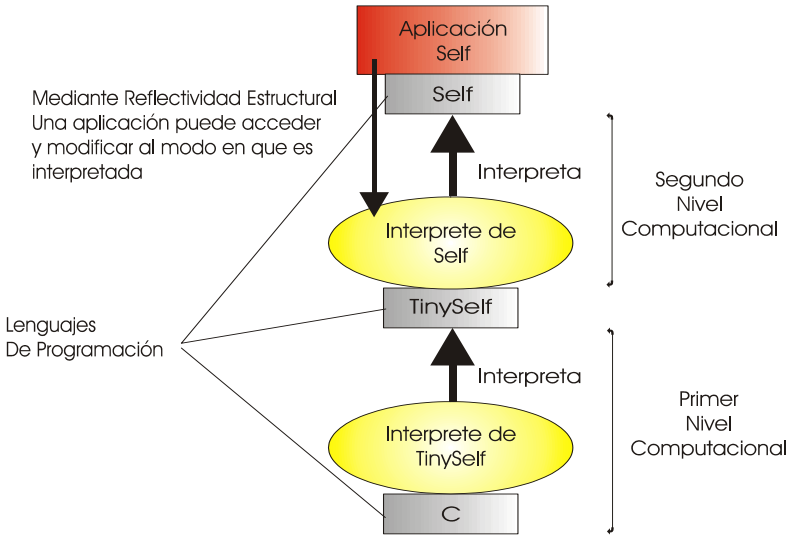

Figura 3.10: Consiguiendo reflexión introduciendo un nuevo intérprete

El resultado es una torre de intérpretes similar a la que se describió anteriormente, donde se obtiene un intérprete de *Self* que permite modificar

determinados elementos de su computación, siempre que éstos hayan sido tenidos en cuenta en el desarrollo del intérprete.

El principal problema es la eficiencia del sistema. El desarrollar dos niveles computacionales (intérprete de intérprete) ralentiza la ejecución de una aplicación codificada en *Self*. Para solventar esto se propone implementar un traductor de código *tinySelf* a otro lenguaje que pueda ser compilado a código nativo [Assumpcao95]. Una vez desarrollado éste traductor, se hará que el código del propio intérprete de *Self* (codificado en el lenguaje *tinySelf*) se traduzca a código nativo, reduciendo así un nivel computacional.

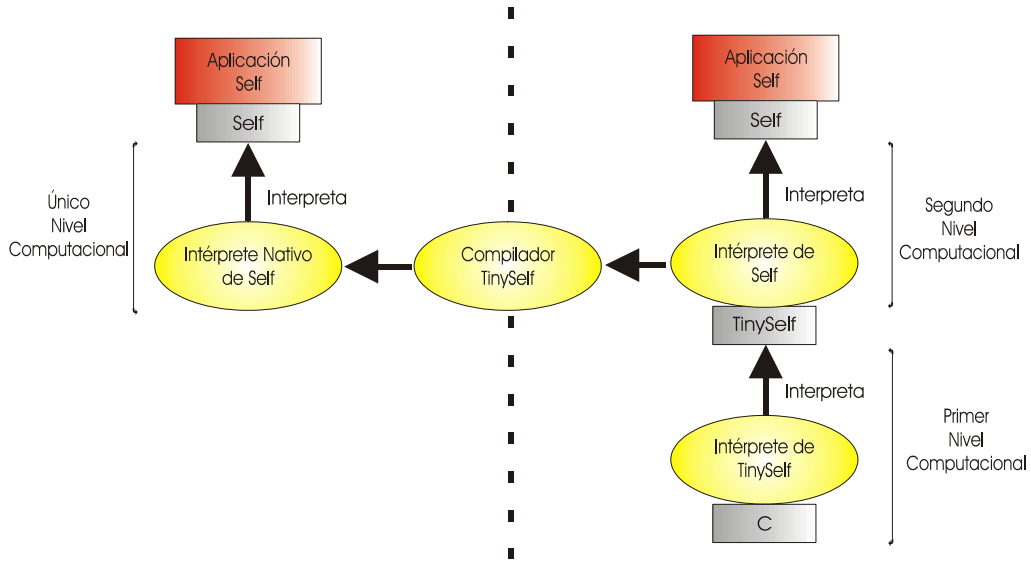

Figura 3.11: Reducción de un nivel de computación en la torre de intérpretes

El resultado se muestra en la Figura 3.11. Con el nuevo esquema, tendremos un único intérprete en lugar de los dos anteriores. Este único intérprete, al ser nativo, alcanzará una eficiencia superior al sistema en dos niveles visto anteriormente. No obstante, el producto final tendría dos inconvenientes:

- Una vez que traduzcamos el intérprete, éste no podrá ofrecer la modificación de una característica que no haya sido prevista con anterioridad. Si queremos añadir alguna, deberemos volver al sistema basado en los dos intérpretes, codificarla, probarla y, cuando no exista ningún error, volver a generar el intérprete nativo, lo cual es un proceso muy lento y complejo.
- La implementación de un traductor de *tinySelf* a un lenguaje compilado es una tarea difícil, puesto que el código generado deberá estar dotado de la información necesaria para ofrecer reflexión estructural en tiempo de ejecución. No obstante, existen sistemas que implementan estas técnicas, como por ejemplo *Iguana* [Gowing96]. Al mismo tiempo, la ejecución de un código que ofrece información modificable dinámicamente (sus objetos) ocupa más espacio, y añade una ralentización en sus tiempos de ejecución.

En un capítulo posterior detallaremos más el modelo de objetos y las capacidades de reflexión de este lenguaje.

3.4.2.3 OBJVLISP

ObjVlisp es un sistema creado como una evolución de *Smalltalk-76* [Ingalls78] y desarrollado como un sistema extensible, capaz de modificar y ampliar determinadas funcionalidades de los lenguajes orientados a objetos [Cointe88]. El modelo computacional del sistema es definido por la implementación de una máquina virtual en el lenguaje *Lisp* [Steele90], máquina que está dotada de un conjunto de primitivas que trabajan con una abstracción principal, el objeto. El modelo computacional del núcleo del sistema se define con seis postulados [Cointe88]:

- Un objeto está compuesto de un conjunto de miembros que pueden representar su información (atributos) y su comportamiento (métodos). La diferencia entre ambos es que los métodos son susceptibles de ser evaluados. Los miembros de un objeto pueden conocerse y modificarse dinámicamente (reflexión estructural).
- La única forma de llevar a cabo una computación sobre un objeto es enviándole un mensaje indicativo del método que queramos ejecutar.
- Al igual que muchos otros lenguajes orientados a objetos, todo objeto ha de pertenecer a una clase que especifique su estructura y comportamiento. Los objetos se crearán dinámicamente como instancias de una clase.
- Una clase es también un objeto instancia de otra clase denominada metaclase. Por lo tanto, aplicando el punto 3, una metaclase define la estructura y el comportamiento de sus clases instanciadas. La metaclase inicial primitiva se denomina `CLASS`, y, para que el modelo propuesto sea coherente, es construida como instancia de sí misma.
- Una clase puede definirse como subclase de otra(s) clase(s). Este mecanismo de herencia permite compartir los miembros de las clases base por sus clases derivadas. El objeto raíz en la jerarquía de herencia se denomina `OBJECT`.
- Los miembros definidos por una clase describen un ámbito global para todos los objetos instanciados a partir de dicha clase, es decir, todos sus objetos pueden acceder a dichos miembros.

A partir de los seis puntos anteriores se deduce que son establecidos, en tiempo de ejecución, dos grafos de objetos en función de dos tipos de asociaciones existentes entre ellos: la asociación "hereda de" (en la que el objeto raíz es `OBJECT`) y la asociación "instancia de" (con objeto raíz `CLASS`). Un ejemplo de ambos grafos se muestra en la siguiente figura:

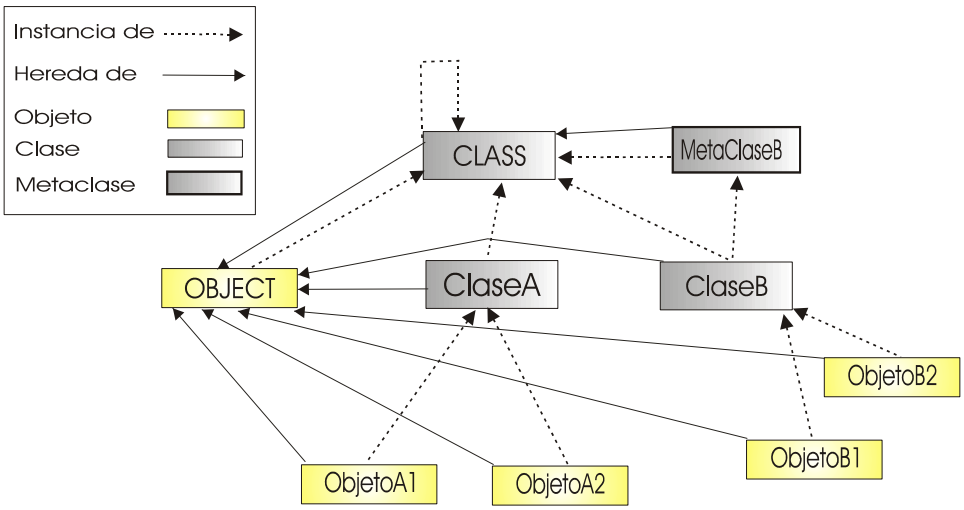

Figura 3.12: Doble grafo de objetos en el sistema *ObjVlisp*

La plataforma permite extender el sistema computacional gracias a la creación dinámica de objetos y a la extensión de los existentes. Los nuevos objetos ofrecen un mayor nivel de abstracción al usuario del sistema. Esto es una consecuencia directa de aplicar la reflexión estructural en tiempo de ejecución.

El sistema soporta un mecanismo de adaptabilidad basado en el concepto de metaclase. Si queremos modificar el comportamiento de los objetos instancia de una clase, podemos crear una nueva metaclase y establecer una asociación entre la clase y ésta, mediante la relación "es instancia de". El resultado es la modificación de parte de la semántica de los objetos de dicha clase. En la Figura 3.12 la semántica de los objetos instancia de la clase B es definida por su metaclase B. Las distintas modificaciones que puede hacer una metaclase en el funcionamiento de los objetos de una clase son:

- Modificación del funcionamiento del mecanismo de herencia.
- Alteración de la forma en la que se crean los objetos, implementando un nuevo comportamiento de la primitiva `make-object`.
- Cambio de la acción llevada a cabo en la recepción de un mensaje.
- Utilización de otro tipo de contenedor para coleccionar los miembros de un objeto.
- Modificación del acceso a los miembros, estableciendo un mecanismo de ocultación de la información.

3.4.2.4 *LINGUISTIC REFLECTION* EN *JAVA*

Según Graham Kirby y Ron Morrison, *Linguistic Reflection* (no confundir con el concepto de reflexión lingüística mencionado antes) es la capacidad de un programa para, en tiempo de ejecución, generar nuevos fragmentos de código e integrarlos en su entorno de ejecución [Kirby98]. Siguiendo este principio, se implementó un entorno de trabajo en el que se añadía al lenguaje de programación *Java* soporte para esta característica. No obstante, se debe tener en cuenta que la capacidad de un sistema reflectivo para adaptarse a un contexto en tiempo de ejecución, modificando su estado, entra en conflicto con el concepto de tipo. Si un objeto tiene un tipo asignado obligatoriamente, en tiempo de compilación será posible conocer exactamente el conjunto de atributos y operaciones de dicho tipo para su uso en el código del programa.

Si pretendemos ampliar o sustituir algún elemento del conjunto de miembros de un objeto, como el tipo de dicho objeto describe exactamente el conjunto de miembros del mismo, incurriríamos en un conflicto, al romper entonces el concepto de tipo (un tipo en tiempo de ejecución no seria exactamente igual a la declaración que existe del mismo en el programa).

Otro problema relacionado con el anterior también ocurre al tratar de modificar los miembros de un tipo. Al tener en la declaración de un tipo un conjunto determinado de miembros asociados, si tratamos de añadir cualquier miembro dinámicamente al mismo no podríamos usarlo en el programa, al no poder tener información de estos miembros añadidos en tiempo de compilación [Cardelli97]: Si tratásemos de crear código para acceder a ellos, el compilador devolverá un error al no poder encontrar en tiempo de compilación una definición válida para el miembro (al introducirse una vez que el programa entra en funcionamiento). El objetivo del prototipo implementado es pues evitar este problema, ofreciendo la generación dinámica de código sin perder el sistema de tipos del lenguaje *Java* [Gosling96].

Como se muestra en la Figura 3.13, los pasos llevados a cabo en la evaluación dinámica de código son:

- La aplicación inicial es ejecutada por una implementación de la máquina virtual de *Java*.
- La aplicación genera código dinámicamente en función de los requisitos propios de su contexto de ejecución.
- Este nuevo fragmento de código es compilado a código binario de la máquina virtual, mediante un compilador de *Java* codificado en *Java*. Este proceso se realiza con la comprobación estática de tipos propia del lenguaje.
- La implementación de una clase derivada de la clase *ClassLoader* [Gosling96] permite cargar dinámicamente el código generado, para pasar a formar parte del entorno computacional de la aplicación inicial.

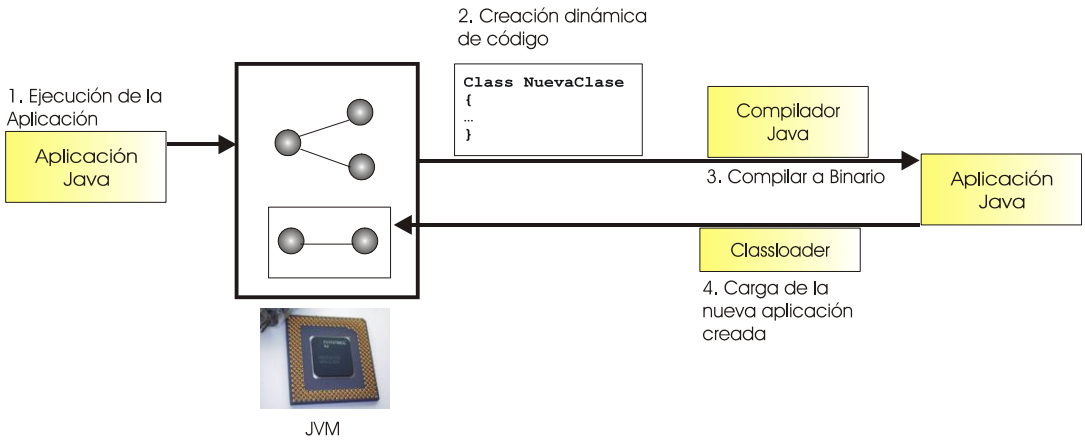

Figura 3.13: Evaluación dinámica de código creado en tiempo de ejecución

La aportación principal del sistema es soportar la codificación de aplicaciones genéricas en tiempo de ejecución y seguras respecto al tipo. Éste es un concepto similar a las plantillas *(template*s) del lenguaje *C++* [Stroustrup98], pero resueltas dinámicamente en lugar de determinar su tipo en tiempo de compilación. Como caso práctico, se han implementado dinámicamente productos cartesianos de tablas de una *BBDD* relacional y contenedores de objetos seguros respecto al tipo, ambos genéricos.

En lo referente a reflexión, el sistema no aporta nada frente a la introspección ofrecida por la plataforma *Java* [Sun97d]. Sin embargo, la posibilidad de crear código dinámicamente ofrece facilidades propias de plataformas dotadas de reflexión estructural como *Smalltalk*. La implementación mediante un intérprete puro [Cueva98] hubiese sido más sencilla, pero menos eficiente en tiempo de ejecución. Por otra parte, se ve cómo un modelo basado en clases plantea dificultades a la hora de hacer modificaciones dinámicas a los tipos de un programa, aspecto que se tratará en profundidad más adelante.

3.4.2.5 PYTHON

Python es un lenguaje de programación orientado a objetos, portable, interpretado y de propósito general [Rossum01]. Es una plataforma libre de código abierto, perteneciendo el *copyright* de su código a la *Python Software Foundation* (*PSF*) [PSF06]. Su desarrollo comenzó en 1990 en el *CWI* de Ámsterdam. El lenguaje posee módulos, paquetes, clases, excepciones, herencia múltiple, manejo automático de memoria y un sistema de chequeo de tipos dinámico. Posee también los conceptos de *duck typing* y metaclases ya mencionados anteriormente. Existen intérpretes del lenguaje en diferentes plataformas como *UNIX*, *Windows*, *Mac,* etc. También podemos destacar múltiples implementaciones de diferentes características de este lenguaje en desarrollo en la actualidad, como *Psyco* [Psyco06] (un compilador *JIT* que convierte código *Python* a código nativo de la máquina especializado de diferentes formas, según los datos manipulados por el programa, para intentar acelerar la ejecución del mismo), *Jython* (un compilador de *Python* a *bytecode Java*) o *IronPython* (traduciendo *Python* a *bytecode* de la plataforma *.NET*) entre otras. Otro proyecto interesante en desarrollo es *PyPy* [PyPy06], consistente en reescribir el lenguaje *Python* usando el propio lenguaje con la intención de crear un sistema altamente optimizado con generación estática de código para diferentes plataformas.

Este lenguaje es dinámico, lo que implica que posee una serie de características particulares de este tipo de lenguajes que se verán en el capítulo posterior destinado a los mismos. Todo código en *Python* se traduce a una representación en forma de *bytecode*, que es ejecutado en la máquina virtual del mismo. Cada vez que un código en *Python* es interpretado, se genera un archivo de *bytecode* (*pyc*) que permite cargar código dinámicamente de una forma más eficiente, ya que este código no será procesado ni traducido. Esto es de especial ayuda cuando se usa un mismo archivo múltiples veces sin que sea sometido a cambios y no durante la ejecución del programa una vez cargado en memoria, ya que el *bytecode* existente en este archivo es equivalente al que el lenguaje genera por el proceso de traducción normal. Si un archivo de código *Python* es alterado, entonces el archivo de *bytecode* asociado se regenera de nuevo.

Una de las características principales de este lenguaje es su extensibilidad. Es posible diseñar y construir fácilmente nuevos módulos que añadan más funciones al mismo, permitiendo diseñar soluciones con este lenguaje adaptadas a múltiples fines. Una de las principales ventajas de este lenguaje es su flexibilidad, debido a las operaciones que permite hacer con sus tipos, y cómo pueden ser tratados y modificados (añadir/modificar/eliminar miembros y relaciones ente objetos) en tiempo de ejecución. En concreto este lenguaje posee reflexión estructural y herencia dinámicas, permitiendo pues que cualquier tipo existente pueda ser ampliado o modificado (tanto en su estructura como en su comportamiento) en tiempo de ejecución y a su vez puedan modificársele sus objetos "padre" cambiando la jerarquía de herencia existente en un momento dado según sea necesario. El modelo de objetos poseído por este lenguaje está basado en el de prototipos. Veremos más detalles acerca del modelo de objetos y sus capacidades reflectivas en un capítulo posterior de esta tesis.

Además de reflexión estructural, el intérprete del lenguaje ofrece numerosa información del sistema en tiempo de ejecución mediante introspección. Sobre la

reflexión estructural del lenguaje se han construido módulos que aumentan las características reflectivas del entorno de programación [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91]. Todas las características reflectivas vistas en este lenguaje son soportadas gracias a la arquitectura del sistema que se presenta en la figura 3.14 [Jackson05]:

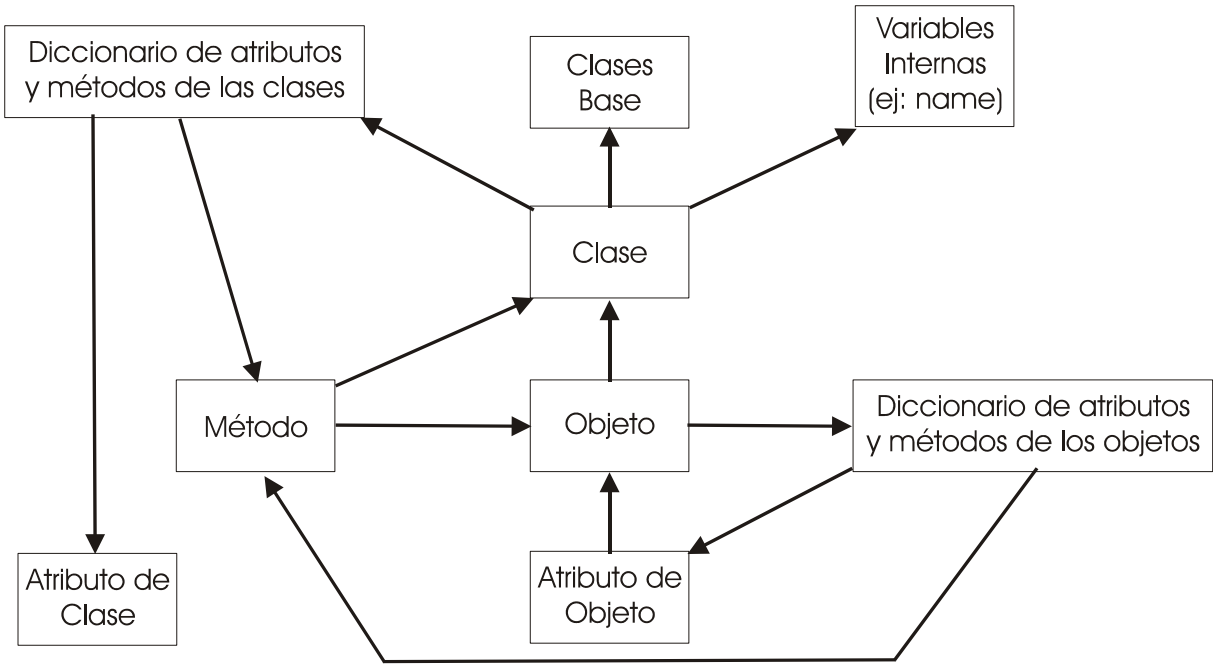

Figura 3.14: Estructura de clases de la arquitectura del lenguaje *Python*

3.4.2.6 RUBY

Ruby [RubyCentral06b] es un lenguaje dinámico, orientado a objetos puro, implementado mediante un intérprete (aunque existen versiones que se compilan a una máquina virtual (*Gardens point Ruby.Net*, versión beta [Gardens06]) y con soporte para reflexión estructural. Principalmente está enfocado a la sencillez y agilidad de desarrollo. Combina una sintaxis basada en *Ada* y *Perl* con capacidades para trabajar con objetos similares a las poseídas por *Smalltalk*. Actualmente existen implementaciones para *Windows*, *Linux*, *Mac OS X* y la plataforma *.NET* [Gardens06]. Es una plataforma libre de código abierto con licencia *GPL* (*General Public License*). Recientemente, la popularidad de este lenguaje se ha visto incrementada gracias al *framework Ruby on Rails* [Thomas05].

Cualquier tipo de dato en *Ruby* es un objeto y toda función es un método. Las variables se construyen como referencias a objetos y no son los propios objetos en sí. Soporta características estándar de lenguajes orientados a objetos (no herencia múltiple) y también permite la programación procedural (aunque de forma simulada, ya que todo procedimiento pertenecerá a un objeto llamado "*Object*", padre de todos los objetos), además de *closures*. En lo referente a sus capacidades reflectivas, *Ruby* soporta introspección [RubyCentral06b] y reflexión estructural (aunque de forma no completa, puesto que se permite modificar la estructura y comportamiento de clases y objetos, pero no permite modificar la clase padre de un objeto dado y por tanto no posee herencia dinámica). Este lenguaje también posee algunos elementos que permiten simular funcionalidades de reflexión computacional (de forma limitada). El modelo computacional de *Ruby* está basado en el modelo de prototipos y utiliza los conceptos de

metaclases y de *duck typing* ya mencionados. En un capítulo posterior se detallarán más las capacidades de reflexión de este lenguaje y las particularidades de su modelo de objetos.

La siguiente versión en desarrollo, *Ruby2* [Davis06] en principio parece que no será compatible hacia atrás y su rendimiento va a tratar de ser mejorado con la utilización de una máquina virtual (*Rite* [Davis06]).

3.4.2.7 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

A lo largo de este capítulo se ha visto cómo los sistemas dotados de reflexión estructural ofrecen una mayor flexibilidad que los que únicamente ofrecen introspección y que este nivel, dadas las facilidades que ofrece, es el que muchos de los lenguajes dinámicos más extendidos implementan. Por ejemplo, mientras que en *C#* (plataforma *.NET*), una aplicación sólo puede modificar su ejecución en función de su estado (conociendo el mismo mediante introspección), en *Smalltalk* además puede modificar su estructura dinámicamente, gracias a la reflexión estructural. Esto provoca diversos beneficios muy importantes:

- **Adaptabilidad del código**: El código de un programa puede cambiar más fácilmente según las circunstancias, el entorno de la aplicación,…
- **Desarrollo de software adaptativo**: Mayor flexibilidad.
- **Un mayor nivel de factorización del código desarrollado**.
- **Soporte para desarrollar *frameworks*** (permite la generación y manipulación de programas de una manera más sencilla).
- También abre una vía para **involucrar al usuario en el desarrollo de aplicaciones** en tiempo de ejecución, donde se vayan cambiando las características de ésta en función de sus preferencias.

Por tanto, vemos como el nivel de flexibilidad de estos sistemas es lo suficientemente elevado como para que cualquier programa pueda reaccionar adecuadamente ante un gran número de posibles cambios a sus requisitos o reglas de negocio, sin necesidad de parar y recompilar la aplicación.

Por último, es destacable que la carencia de reflexión computacional de estos sistemas puede limitar sus aplicaciones en ciertos entornos, donde se requiera una flexibilidad aún más elevada. No obstante, la flexibilidad que ofrece la reflexión estructural es suficiente para la mayoría de los posibles casos que se puedan plantear, a tenor del uso que se está dando a la misma actualmente con los lenguajes que la soportan (actualmente podemos ver lenguajes muy extendidos que incorporan este nivel de reflexión, como *Python,* y que se han usado para el desarrollo de productos comerciales con notable éxito, como *Zope* [Zope06], por citar un ejemplo significativo).

Además existe un factor cuya importancia no es desdeñable, y es que es razonable suponer que el coste de incorporar reflexión estructural a un sistema es menor que el coste de incorporar reflexión computacional, debido a la menor complejidad de la primera, que se traduciría en un menor número de operaciones adicionales (y un mayor rendimiento) mientras los programas estén en ejecución. La complejidad de los sistemas que implementan reflexión computacional se estudiará posteriormente.

3.4.3 Reflexión en Tiempo de Compilación

Basándonos en la clasificación realizada en el capítulo anterior, la reflexión en tiempo de compilación se produce siempre en fase de traducción. En este tipo de sistemas, una aplicación puede adaptarse a un contexto dinámicamente siempre y cuando los cambios que se vayan a hacer hayan sido contemplados en su código fuente, ya que toda su información relativa a estos cambios se generaría en tiempo de compilación. Por tanto, el principal problema de esta técnica es que si en tiempo de ejecución aparecieren requisitos no previstos en fase de desarrollo del sistema, éstos no podrían solventarse dinámicamente, ya que la aplicación debería recodificarse y recompilarse, no pudiendo responder por tanto a estos cambios mientras el programa está en funcionamiento y obligándonos a detener el proceso que realiza.

3.4.3.1 OpenC++

OpenC++ [Chiba95] es un lenguaje de programación que ofrece características reflectivas a los programadores de *C++*. Su característica principal es la eficiencia dinámica de las aplicaciones creadas, ya que no existe una sobrecarga computacional en tiempo de ejecución debido a que toda característica reflectiva se procesa en tiempo de compilación. Este lenguaje está enfocado a amoldar el lenguaje de programación y su semántica al problema a tratar, para así poder desarrollar las aplicaciones a su nivel de abstracción adecuado.

El modo en el que son generadas las aplicaciones se basa en un preprocesado del código fuente. Una aplicación cualquiera se codifica en *OpenC++,* pudiendo hacer uso de sus características reflectivas. Ésta aplicación es traducida a código *C++* que, tras compilarse mediante cualquier compilador estándar, genera la aplicación final, adaptable en el grado estipulado cuando fue codificada. La arquitectura presenta el concepto de metaobjeto popularizado por Gregor Kiczales [Kiczales91] para modificar la semántica determinados elementos del lenguaje. En concreto, para las clases y funciones miembro, se puede modificar el modo en el que éstas son traducidas al código *C++* final.

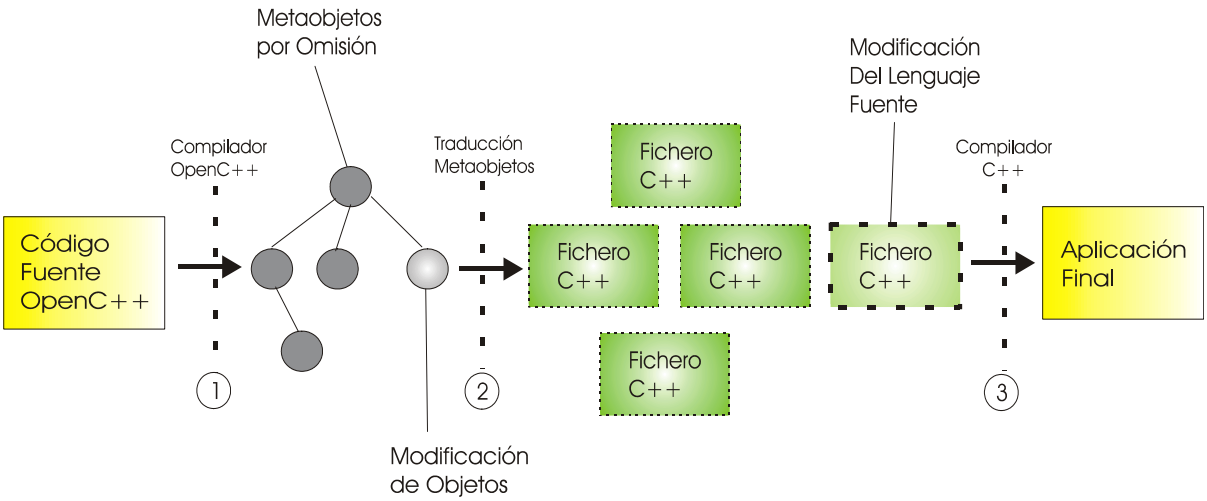

Figura 3.15: Fases de compilación de código fuente *OpenC++*

Tomando código *OpenC++* como entrada, el traductor de este lenguaje genera *C++* estándar como salida. Este proceso consta de 2 fases, mostradas en la Figura 3.15.

En la primera fase, se crea el árbol sintáctico relativo a las clases y funciones miembros de la aplicación, creándose un metaobjeto por cada uno de dichos elementos sintácticos. Los metaobjetos serán instancias de una clase `Class` (si el elemento sintáctico es una clase), o bien de una clase `Function` (cuando se trate de un método). Estas dos clases por defecto traducen el código sin añadirle ninguna modificación y serán las asignadas a los elementos sintácticos por defecto. Si una clase o método va a ser definidos mediante un metaobjeto en *OpenC++*, se crearán para ello instancias de clases derivadas de las dos mencionadas anteriormente. Dichas clases derivadas dictarán la forma en la que se traducirá el código fuente asociado a los metaobjetos. Ésta segunda fase de traducción concluye con la generación de la aplicación final por un compilador de *C++*.

El sistema ofrece un mecanismo de preproceso para poder amoldar el lenguaje de programación *C++* [Stroustrup98] a las necesidades del programador. Además de poder modificar parte de la semántica de métodos y clases (reflexión computacional), *OpenC++* ofrece la posibilidad de añadir palabras reservadas al lenguaje, para implementar modificadores de tipos, clases y el operador `new` (reflexión lingüística).

3.4.3.2 OPENJAVA

OpenJava es una ampliación de las características reflectivas de *Java* enfocada únicamente a actuar en tiempo de compilación [Chiba98]. Como ya se ha mencionado, el lenguaje *Java* ofrece introspección por medio de la clase `Java.lang.Class` [Sun97d], entre otras. Mediante la creación de una nueva clase (`OJClass`) y el preproceso de código *OpenJava* realizado por el *OpenJava compile*r, se diseñó un lenguaje capaz de ofrecer reflexión en tiempo de compilación que ofrece las siguientes características:

- Modificación del comportamiento de determinadas operaciones sobre los objetos, como invocar métodos o acceder a atributos (reflexión computacional restringida).
- Reflexión estructural. Se añaden a objetos y clases funcionalidades propias de reflexión estructural, de forma conjunta a la introspección ya ofrecida por *Java*.
- Modificación de la sintaxis del lenguaje. Pueden crearse nuevas instrucciones, operadores y palabras reservadas (reflexión lingüística).
- Modificación de los aspectos semánticos del lenguaje. Es posible modificar la promoción o coerción de tipos [Cueva95b].

OpenJava mejora las posibilidades ofrecidas por su "hermano" *OpenC++*, sin necesidad de modificar la implementación de la máquina abstracta, como se hace en otras implementaciones (ver *MetaXa* [Kleinöder96]). Esto supone dos ventajas:

- Al no modificarse la máquina virtual, no se generan distintas versiones de ésta con la consecuente pérdida de portabilidad de su código fuente [Ortin01].
- Al no ofrecer adaptabilidad dinámica, no supone una pérdida de eficiencia en tiempo de ejecución.

El lenguaje de programación *OpenJava* se ha utilizado para describir la solución de problemas mediante patrones de diseño [GOF94] en el nivel de abstracción adecuado [Tatsubori98]; el lenguaje se modificó para amoldar su sintaxis a los patrones *adapter* y *visitor* [GOF94].

3.4.3.3 JAVA DYNAMIC PROXY CLASSES

En la edición estándar de la plataforma *Java2 (Java Standard Editio*n), a partir de la versión 1.2.3, se ha aumentado la funcionalidad ofrecida por el paquete `java.lang.reflect` para dotarle de un mecanismo de modificación del paso de mensajes de una determinada clase. A la clase que permite este tipo de operaciones se la denomina clase *proxy* (apoderado) [Sun99].

Una clase *proxy* especifica la implementación de un conjunto ordenado de *interfaces*. La clase es creada dinámicamente especificando su manejador de invocaciones (`InvocationHandler`). Cada vez que se invoque a un método de las *interfaces* implementadas, la clase apoderada ejecutará el método `invoke` de su manejador, que podrá modificar su semántica en función del contexto dinámico existente. El diagrama de clases que ofrece este marco de trabajo se muestra a continuación:

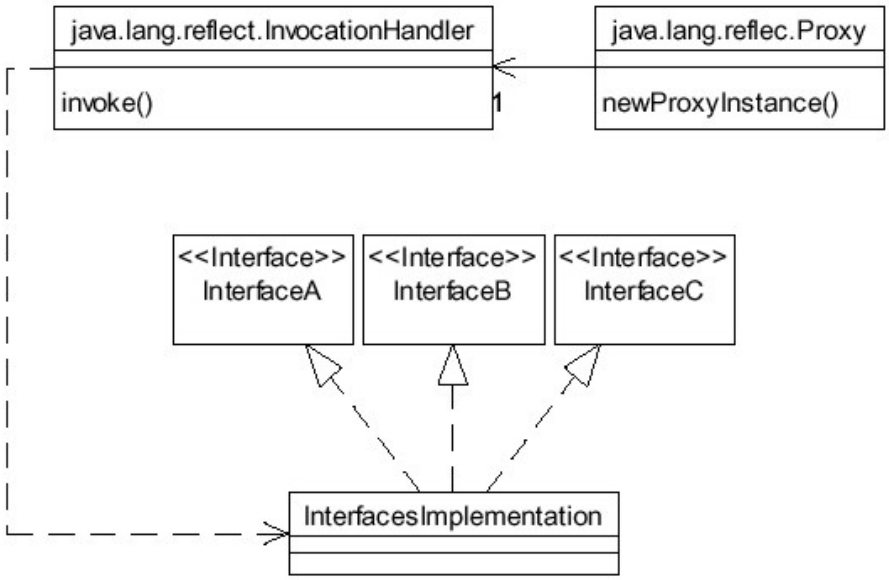

Figura 3.16: Diagrama de clases de la implementación de un manejador de invocaciones [Ortin01]

La clase es creada dinámicamente especificando las interfaces a implementar, un cargador de clases (*ClassLoader* [Gosling96]) y una implementación del manejador de invocaciones (*InvocationHandler*). Esta funcionalidad se ofrece a través del método de clase *newProxyInstance* de la clase *Proxy*.

La adaptabilidad del sistema se centra en la implementación del método *invoke* del manejador. Este método recibe el objeto real al que se ha pasado el mensaje, el identificador del método y sus parámetros. Haciendo uso del sistema introspectivo de la plataforma *Java* [Sun97d], podremos conocer e invocar el método que deseemos en función de los requisitos existentes en tiempo de ejecución. Con este mecanismo, la plataforma de *Java* ofrece un grado de flexibilidad superior a la introspección existente en versiones anteriores, al permitir modificar la semántica del paso de mensajes. Sin embargo, las distintas interpretaciones de dicha semántica han de especificarse en tiempo de compilación para ser seleccionadas posteriormente de forma dinámica, limitando la flexibilidad del conjunto final.

En resumen, lo que se tiene es un conjunto de clases que, haciendo uso de la introspección de la plataforma, permite simular la modificación de una parte del

comportamiento del sistema. Sin embargo, no estamos ante una implementación que permita tener reflexión computacional completa, al no permitir la modificación de dicho comportamiento para todo el sistema.

3.4.3.4 HASKELL

Haskell es un lenguaje puramente funcional de propósito general. Además del soporte que *Haskell* ya posee para introspección en tiempo de ejecución mediante su módulo *Data.Generic*, también existen algunas modificaciones relativas a la reflexión estructural. En concreto existe una implementación que hace una aportación significativa a la hora de permitir que *Haskell* haga una modificación de la estructura de los elementos de un programa. Dicha modificación ocurriría en tiempo de compilación y es denominada *Template Haskell* [Lynagh03]. Esta implementación consiste en una extensión del lenguaje *Haskell98* [Haskell05] que permite hacer metaprogramación en tiempo de compilación mientras se mantiene la seguridad de tipos. Este mecanismo permite la conversión de sintaxis concreta de *Haskell* en árboles sintácticos abstractos (*AST*), árboles que podrán ser manipulados en tiempo de compilación por código *Haskell* y que están representados por tipos de datos abstractos. El mecanismo seguido es bastante simple conceptualmente: Dado un código en *Haskell* concreto, en tiempo de compilación dicho código podrá ser cosificado y convertido en un árbol sintáctico abstracto. Esta estructura podrá después ser manipulada y convertida luego de nuevo en código *Haskell* concreto, "pegando" el nuevo código generado de esta forma a las partes del programa donde se especifique.

Por el momento el único compilador que soporta estas extensiones es *GHC* [Marlow05], aunque todavía es una implementación incompleta que no soporta todas las características del modelo teórico.

3.4.3.5 JAVASSIST

Se menciona aquí *Javassist* (*Java Programming Assistant*) [Chiba06] por ser una librería que sirve de base para construir sistemas que soporten capacidades reflectivas y también capacidades orientadas a aspectos. Esta librería es un subproyecto del proyecto *JBoss* [Jboss06], que permite manipular los *bytecode* de *Java* de forma simple, de manera que permita incluso la edición directa de los mismos. De esta forma se puede definir una nueva clase en tiempo de ejecución y también se puede modificar el archivo de una clase cuando ésta es cargada por la máquina virtual *JVM*. Existen además dos niveles de funcionamiento:

- El nivel de código, que permite modificar el archivo que contiene la clase sin saber nada acerca de la especificación de los *bytecode* de *Java,* trabajando sólo con la sintaxis de dicho lenguaje.
- El nivel de *bytecode*, donde se permite la edición de archivos fuente de *Java* modificando sus *bytecodes* directamente.

Esta librería puede ser usada para hacer tareas como:

- Programación orientada a aspectos: *Javassist* puede ser un medio para añadir nuevos métodos a una clase, o la creación de código que se ejecutaría antes o

después de un método, algo que es usado en programación orientada a aspectos.

- Reflexión: Mediante el uso de *Javassist* se podría lograr reflexión de forma limitada en tiempo de ejecución, permitiendo a los programas usar un metaobjeto que controle las llamadas a otros objetos modificando su comportamiento al realizarlas, sin necesidad de cambiar el compilador o máquina virtual.

No obstante se ha incluido *Javassist* en esta sección debido a que los cambios en las clases no se pueden hacer dinámicamente. Si se cambia el archivo que contiene la clase, al cargarse dicha clase en memoria por la máquina virtual, los cambios realizados se verán adecuadamente reflejados. No obstante, una vez cargada dicha clase en memoria no se podrá "descargar" para hacer más modificaciones que puedan quedar patentes en la ejecución posterior, por lo que la funcionalidad que proporciona no se puede catalogar de dinámica. Mediante el uso de esta librería se han implementado soluciones para una serie de problemas, como por ejemplo el sistema *Reflex* [Reflex06], que presenta una solución para implementar orientación a aspectos dinámica usando un mecanismo de reflexión computacional parcial. Una descripción completa del uso de esta librería se puede encontrar en [Chiba06b].

3.4.3.6 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

La principal carencia de los sistemas reflectivos en tiempo de compilación es la incapacidad de poder amoldarse a situaciones no previstas en fase de desarrollo. Toda la adaptabilidad de un sistema deberá definirse cuando éste es construido, ya que una vez que esté en ejecución no podrá adecuarse de otra manera. Por otro lado, al determinarse el grado de flexibilidad en tiempo de compilación, estos sistemas normalmente ofrecen un alto rendimiento en tiempo de ejecución, superior a los adaptables dinámicamente.

3.4.4 Reflexión y Programación Orientada a Aspectos

Se ha incluido este paradigma en esta sección debido a que mediante el uso de los elementos definidos en él se puede lograr reflexión estructural. Gregor Kiczales [Kiczales97], Chris Maeda y su equipo de *Xerox PARC* [Xerox05] son los que iniciaron este paradigma, desarrollando la primera herramienta orientada a aspectos, que por otra parte es la más extendida hoy en día, *AspectJ* [AspectProgramming06] [AspectJ06].

La *POO* es hoy en día la metodología que se utiliza mayoritariamente en los nuevos proyectos de desarrollo de *software*. La *POO* ha demostrado su fuerza a la hora de modelar la funcionalidad básica dominante del sistema (normalmente la funcionalidad original para la que surge la aplicación), pero no es capaz de tratar de forma adecuada otros aspectos que no forman parte de esta funcionalidad básica o dominante, ya que éstos suelen encontrarse diseminados por los diferentes módulos de los que esta compuesto el *software*, a menudo sin relación entre ellos, con el consiguiente problema de pérdida de claridad en el código (lo que afecta negativamente a su calidad). Se puede afirmar por lo tanto que las técnicas tradicionales no gestionan bien la "separación de incumbencias" para aspectos que no forman parte de la funcionalidad básica del sistema [Tarr99].

El principio **de la separación de incumbencias**[5] (*Separation of Concerns* o *SoC*) [Parnas72] [Dijkstra76] [Hürsch95] separa los algoritmos principales de una aplicación de aquellas incumbencias con un propósito especial (típicamente ortogonal a la funcionalidad principal), construyendo las aplicaciones finales mediante la unión del código de su funcionalidad principal más el de sus incumbencias de dominio específico. Su objetivo es conseguir que los distintos componentes de una aplicación puedan ser localizados en el diseño y en el código, y que puedan ser tratados explícitamente. De esta forma cada función del sistema podrá ser fácilmente comprendida, modificada, añadida y reutilizada. Los principales beneficios de la *SoC* son:

- Un nivel de abstracción más alto, debido a que el desarrollador se puede centrar en incumbencias concretas y de forma aislada.
- Una mayor facilidad a la hora de entender la funcionalidad de la aplicación. El código que implementa ésta no está entremezclado con código de otras incumbencias.
- Una mayor reusabilidad al haber un menor acoplamiento.
- Una menor complejidad del código resultante.
- Una mayor flexibilidad en la integración de componentes.
- Un incremento de la productividad en el desarrollo.

Un ejemplo de este tipo de incumbencias son los servicios de *log*. En un mismo sistema pueden existir múltiples partes que necesiten guardar un histórico de las acciones que realizan, por lo que un servicio de *log* puede afectar a varias partes del mismo (múltiples clases o métodos). Modelar este tipo de incumbencias mediante un modelo basado en clases tradicional, de manera que queden completamente separadas de la aplicación sobre la que actúan y que sean reutilizables por otros programas sin efectuar cambios a las mismas, es una tarea que la *POO* tradicional no puede resolver satisfactoriamente. Por tanto, los aspectos permiten un nivel de abstracción superior al que permiten las clases y objetos, al estar pensados para modelar de esta forma incumbencias como la descrita.

La programación orientada a aspectos (*Aspect Oriented Programming* o *POA*) es una de las aproximaciones a este principio. La *POA* ofrece soporte directo en el lenguaje para modularizar incumbencias que cortan transversalmente al código de la funcionalidad básica de la aplicación; separando este código del de los aspectos que lo cortan, el código de la aplicación no estará entremezclado, siendo más fácil de mantener, depurar y modificar.

Gran parte de los sistemas que ofrecen *POA* son estáticos: una vez que la aplicación se ha generado no se puede adaptar en tiempo de ejecución. Sin embargo, hay veces que es necesario poder adaptar una aplicación en ejecución en respuesta a cambios del entorno, o por que surjan nuevos requerimientos cuando el sistema está ejecutándose (y puede ocurrir que no sea posible detenerlo). Por ello, también existen una serie de sistemas que ofrecen *POA* de forma dinámica, pero en muchas ocasiones restringen la forma en que se pueden adaptar las aplicaciones, y muchos de ellos en realidad no son realmente dinámicos.

La mayoría de los sistemas existentes (ya sean dinámicos o estáticos) presentan una restricción muy importante: la dependencia del lenguaje (sólo se puede utilizar una

5 *También es traducida por Separación de Competencias. En este texto se utilizarán indistintamente los términos incumbencia y competencia.*

lenguaje fijado por el sistema). Sin embargo, actualmente existen líneas de investigación que tratan de eliminar esta restricción [Vinuesa07].

Por tanto, la Programación Orientada a Aspectos dota al sistema de la capacidad de modularizar aspectos que se diseminan a través de la funcionalidad básica. Los aspectos expresan funcionalidad que afecta al sistema de una manera ortogonal a la funcionalidad básica, y permiten al desarrollador diseñar el sistema básico sin la necesidad de incluir los aspectos ortogonales, dotándole además de un punto único para realizar modificaciones. La separación del código que contiene la funcionalidad básica del código de los aspectos ortogonales permite que el código resultante no esté entremezclado, siendo más fácil de depurar, mantener y modificar [Parnas72]. Es decir se pretende poder implementar un sistema de forma eficiente y fácil, lo que redundará en una mejor calidad del *software*.

3.4.4.1 PROGRAMACIÓN ORIENTADA A ASPECTOS: DEFINICIÓN

Como ya se ha mencionado, existen situaciones en las que los lenguajes orientados a objetos no permiten modelar de forma suficientemente clara las decisiones de diseño tomadas previamente a la implementación. El sistema final se codifica entremezclando el código propio de la especificación funcional del diseño con llamadas a rutinas de diversas librerías encargadas de obtener una funcionalidad adicional (por ejemplo, distribución, persistencia o multitarea). El resultado es un código fuente excesivamente difícil de desarrollar, de entender, y por lo tanto de mantener [Kiczales97].

En la *POA* hay dos términos muy importantes:

- **Un componente** (*component*) es aquel módulo *software* que puede ser encapsulado en un procedimiento (un objeto, método, procedimiento o *API*). Los componentes serán unidades funcionales en las que se descompone el sistema.
- **Un aspecto** (*aspect*) es aquel módulo *software* que no puede ser encapsulado en un procedimiento. No son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan la ejecución o semántica de los componentes. Ejemplos de aspectos son la gestión de la memoria, la sincronización de hilos (*threads*) o los servicios de *log* antes mencionados.

Una aplicación orientada a aspectos es el resultado de adaptar (modificar o ampliar el funcionamiento) los componentes de una aplicación por medio de aspectos. Este proceso se denomina tejido (*weave*) y es realizado por el tejedor de aspectos (*aspect weaver*). En la Figura 3.17 se puede ver el esquema de funcionamiento de una aplicación orientada a aspectos.

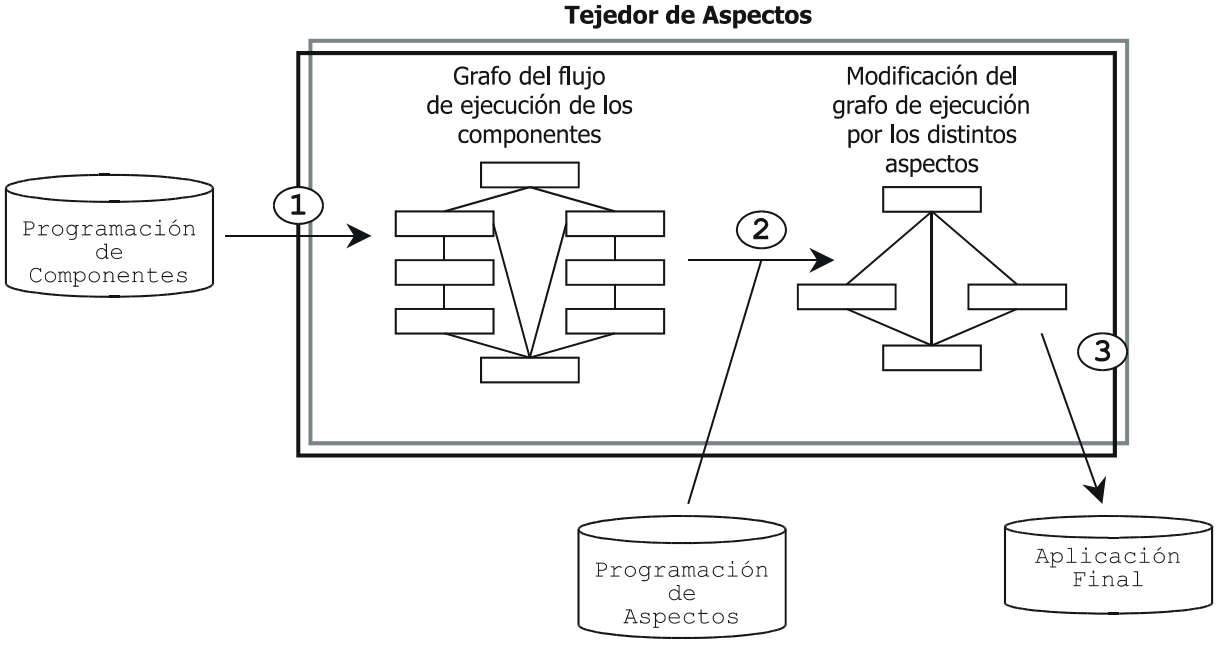

Figura 3.17: Esquema de la Programación Orientada a Aspectos

La secuencia de pasos que se sigue es:

- El tejedor construye un grafo del flujo de ejecución de los componentes del programa (esto lo puede hacer a partir del código fuente o de código ya compilado, depende del sistema).
- A continuación se modifica el grafo anterior realizando las modificaciones o adiciones oportunas (en respuesta a los aspectos).
- Por último se genera el código final de la aplicación (a partir del nuevo grafo generado, que incluye la funcionalidad de los componentes y de los aspectos).

En la actualidad se está realizando mucho trabajo de investigación sobre esta tecnología y han aparecido muchos sistemas que la soportan, siendo uno de los más importantes hasta el momento *AspectJ* [AspectJ06], el cual ha sido utilizado en grandes proyectos con buen resultado y con el que posteriormente veremos un ejemplo que se relaciona con el tema tratado en esta tesis. La mayoría de los otros sistemas que ofrecen *SoC* existentes están evolucionando para ser compatibles y complementarios con la *POA* (por ejemplo la programación adaptable con el sistema *Demeter AspectJ* [DAJ06]).

3.4.4.2 Funcionamiento de la Programación Orientada a Aspectos:

Básicamente en el funcionamiento de la *POA* se pueden identificar tres pasos:

- **Descomposición en aspectos**: se descomponen los requerimientos con el fin de identificar las distintas incumbencias (las comunes y aquéllas que se entremezclan con el resto).

- **Implementación de incumbencias**: se implementan de forma independiente las incumbencias detectadas anteriormente, tanto la funcionalidad básica como los aspectos ortogonales.
- **Recomposición de aspectos**: este proceso, denominado tejido (*weaving*), toma todos los módulos implementados anteriormente (funcionalidad básica y aspectos) y los teje para formar el sistema completo. Es realizado por el tejedor de aspectos (*aspect weaver*).

Los lenguajes orientados a aspectos definen una nueva unidad de programación de *software*, el aspecto, para encapsular las funcionalidades que están diseminadas y enmarañadas (*crosscut* y *tangled*) por todo el código. A la hora de formar el sistema se ve que hay una relación entre los componentes y los aspectos, y que, por lo tanto, el código de los componentes y de estas nuevas unidades de programación tiene que interactuar de alguna manera. Para que los aspectos y los componentes se puedan mezclar, deben tener fijados los puntos donde puedan hacerlo, que son lo que se conoce como puntos de enlace (*joinpoints*). Los puntos de enlace son una interfaz entre los aspectos y los módulos de componentes que define en qué lugares se puede aumentar el comportamiento de los módulos con el comportamiento de los aspectos.

Como ya hemos visto, el proceso de realizar esta unión se conoce como tejido (*weave*), y el encargado de realizarlo es el tejedor de aspectos (*aspect weaver*). El tejedor, además de los aspectos, los módulos y los puntos de enlace, recibe unas reglas que le indican cómo debe realizar el tejido. Estas reglas son los llamados puntos de corte (*pointcuts*), que indican al tejedor qué aspecto tiene que aumentar a qué módulo a través de qué punto de enlace.

El código de los aspectos normalmente recibe el nombre de *advice* al ser el término utilizado en el sistema *AspectJ* [AspectJ06] y que han adoptado la mayoría de sistemas posteriores. Realmente los aspectos describen unos añadidos al comportamiento de los objetos, hacen referencia a las clases de los objetos y definen en qué punto se han de colocar los añadidos.

En las metodologías tradicionales, el proceso de generar un programa consistía en pasar el código a través de un compilador o un intérprete para así disponer de un ejecutable. En la *POA* no se tiene un único código del programa, sino que está separado el código que implementa la funcionalidad básica del código que implementa cada uno de los aspectos. Todo este código debe pasar no sólo a través del compilador, sino que debe ser tratado por el tejedor, que es el que se encarga de crear un único programa con toda la funcionalidad, la básica más los aspectos.

3.4.4.3 TERMINOLOGÍA Y CONCEPTOS DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS

Explicaremos a continuación con algo más de detalle algunos de los conceptos relacionados con *POA* ya mencionados en el punto anterior, pero que requieren alguna explicación adicional.

ASPECTO

Puesto que la *POA* es una aproximación de la *SoC*, antes de definir un aspecto es mejor saber qué es una incumbencia o competencia (*concern*). *"Incumbencia es todo aquello que incumbe al software"* [Tarr99]. Básicamente incumbencia es todo lo que sea

importante para la aplicación, ya sea código, infraestructura, requerimientos, elementos de diseño, etc.

Un aspecto es una clase particular de incumbencia. La definición formal más aceptada es: *"Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa"* [Kiczales97].

De una forma más básica se puede decir que los aspectos son elementos que se diseminan por todo el código y son difíciles de describir con respecto a otros componentes, es decir, módulos *software* que no pueden ser encapsulados en un procedimiento. Los aspectos no son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan a la ejecución o semántica de los componentes.

La estructura de un programa orientado a aspectos se muestra en la Figura 3.18. Se puede ver que el programa es una combinación de distintos módulos. Unos contienen la funcionalidad básica (modelo de objetos), mientras que los demás recogen otro tipo de características como son la seguridad, persistencia, *logging*, gestión de memoria, etc.

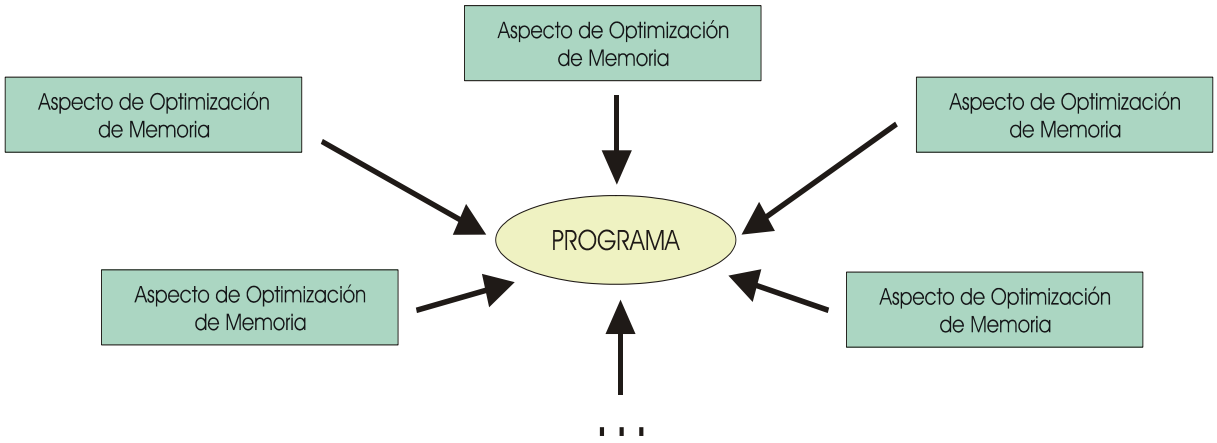

Figura 3.18 Estructura de un programa orientado a aspectos

JOIN POINTS Y *POINTCUTS*

Un punto de corte (*pointcut*) es un conjunto de instrucciones que se le pasan al tejedor con el fin de que sepa qué código (*advice* del aspecto) se le debe añadir en qué punto de enlace a una aplicación. Un ejemplo podría ser invocar el método X del aspecto cuando se produzca un acceso de lectura al campo Y de la clase C. Para que un aspecto sea útil, el desarrollador debe especificar exactamente dónde desea que estos *advices* sean conectados. Como ya hemos mencionado, los posibles destinos para hacer esta conexión son denominados puntos de unión (*join points*), y estarán bien definidos a lo largo de la ejecución de un programa. No todos los puntos de ejecución de un programa son puntos de unión, sólo aquéllos que puedan ser tratados de forma controlada. Cada sistema que soporta *POA* ofrece una serie de puntos de enlace que pueden ser distintos, como por ejemplo[6]:

6 A modo de ejemplo, se muestran los puntos de enlace de AspectJ.

- Invocación a un constructor.
- Ejecución de un método.
- Ejecución de un constructor.
- Inicialización de objetos que se crean con el constructor.
- Preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (`super`).
- Inicialización de bloque estático.
- Acceso de lectura a campo.
- Acceso de escritura a campo.
- Cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque `catch`.
- Ejecución de cualquiera de los *advice* inyectados.

Mediante esta técnica los desarrolladores pueden ampliar la funcionalidad, uniendo código de cualquier clase ya existente o desarrollada en un futuro. Por ejemplo, un *pointcut* puede estar formado por todas las llamadas a una serie de métodos que cumplan con unas características determinadas por el programador. Por tanto, el conjunto de puntos de unión soportados, y la expresividad del lenguaje de especificación de puntos de corte, son una forma de diferenciar los diversos sistemas que soportan *AOP*. Algunos sistemas usan sintaxis basada en *Java*, como *AspectJ* [AspectJ06], mientras que otros como *AspectWerkz* [AspectWerkz06] [Vasseur04], usan una sintaxis más próxima al *XML*.

WEAVING

El proceso de *weaving* (inyección del código de los aspectos en los puntos de unión asociados con este código) es el punto más importante para cualquier solución que emplee *AOP*. En la definición original de *AOP* [Kiczales97], Kiczales y su equipo especificaron las siguientes posibilidades para hacer *weaving*:

- Un preprocesado de código que haga las sustituciones pertinentes al mismo.
- Un post-procesador que "parchee" archivos binarios.
- Un compilador que soporte *AOP* y que genere ya archivos con el proceso de *weaving* realizado.
- *Weaving* en tiempo de carga, haciendo el proceso de *weaving* cuando las clases son cargadas en memoria, como haría *Java* con la *JVM*.
- *Weaving* en tiempo de ejecución, capturando cada punto de unión mientras el programa está en funcionamiento y ejecutando el código que corresponda.

Las dos primeras opciones complican el desarrollo del programa, mientras que las dos últimas tendrán un más que probable impacto en la eficiencia del programa. Adicionalmente, el hacer este proceso en tiempo de ejecución también requiere un entorno preparado para el tratamiento de aspectos, como por ejemplo una *JVM* modificada si trabajamos en entornos *Java*.

Todas las soluciones para hacer *weaving* vistas anteriormente, salvo las dos

últimas, implican cambiar el código de alguna manera. El código que el compilador genera para una determinada clase *Java* después de procesarlo y/o cargarlo en memoria no es el mismo que lo que generaría un compilador estándar de *Java*, ya que ahora contendría "piezas" de código unidas.

Cohen y Gil [Cohen04] han descrito una alternativa, presentando el concepto de "*weaving* en tiempo de despliegue" (*deploy-time weaving*). Este concepto implica un postprocesamiento del código pero, en vez de procesar el código generado, lo que se generan son subclases de las clases existentes, introduciendo las modificaciones pertinentes mediante sobreescritura de métodos. Las clases que ya existían no se modifican en ningún momento y todas las herramientas ya existentes pueden ser usadas durante el desarrollo. Una aproximación similar a ésta se ha usado en la implementación de servidores de aplicaciones *J2EE* como *IBM WebSphere*.

Una de las herramientas más populares que soportan *AOP* es *AspectJ* [AspectJ06], que puede ser empleada para lograr funcionalidades de reflexión estructural en tiempo de compilación [AspectProgramming06] [AspectProgramming06b]. *AspectJ* es una extensión del lenguaje de programación *Java* orientada a aspectos. La versión original de esta herramienta usaba el compilador para efectuar todo el proceso involucrado en el manejo de aspectos. El compilador genera archivos *.class* de *Java* estándar, que pueden ser ejecutados por cualquier *JVM*. Fruto de su fusión con *AspectWerkz* [AspectWerkz06] para formar un único sistema, es de esperar que incorpore a su vez soporte para *AOP* en tiempo de carga. Todo el trabajo del grupo *Xerox* relacionado con aspectos fue integrado en el *IDE* de *Java Eclipse*, de la fundación *Eclipse* [Eclipse06], en diciembre del 2002, lo que ha hecho que *AspectJ* sea una de las herramientas orientadas a aspectos más usada. Otros *frameworks* comerciales que utilizan orientación a aspectos son *Jboss* [Jboss06] y *Spring AOP* [SpringAOP06].

3.4.4.4 PROGRAMACIÓN ORIENTADA A ASPECTOS DINÁMICA

La programación orientada a aspectos se puede clasificar según varios criterios. Si se utiliza como criterio el momento en el que se realiza el tejido, se distinguen tejido estático y tejido dinámico. Hasta ahora hemos visto sistemas que soportan el tejido estático solamente. Describiremos a continuación, a modo de ejemplo, un conjunto de sistemas que permiten hacer este proceso de tejido de forma dinámica y su funcionamiento general.

TEJIDO DINÁMICO

Usando un tejedor estático, el programa final se genera tejiendo el código de la funcionalidad básica y el de los aspectos seleccionados en la fase de compilación. Si se quiere enriquecer la aplicación con un nuevo aspecto, o incluso eliminar uno de los aspectos actualmente tejidos, el sistema debe ser recompilado y reiniciado de nuevo.

Aunque no todas las aplicaciones necesitan ser adaptadas mediante aspectos en tiempo de ejecución, hay aspectos específicos que se benefician de un sistema con tejido dinámico; puede haber aplicaciones que necesiten adaptar sus competencias específicas en respuesta a cambios en el entorno de ejecución [Popovici01]. Ejemplos de ello son las técnicas relacionadas que se han empleado en gestionar requerimientos de calidad del servicio en sistemas distribuidos de *CORBA* [Zinki97], en la gestión de sistemas de "*cache prefetching*" (precarga en memoria) en un servidor *Web* [Segura-Devillechaise03], y en distribución de incumbencias basada en el balanceo de carga [Matthijs97]. Otro ejemplo de uso de *POA* de forma dinámica es el denominado *software* autónomo (*autonomic software/computing*): *software* capaz de auto repararse,

gestionarse, optimizarse o recuperarse [Autonomic06].

Para usar el tejido dinámico, en un primer paso un programa escrito en un lenguaje normal pasa por el compilador y se genera un ejecutable que se pone en ejecución (éste es el proceso tradicional). Este programa no tiene por qué tener ningún conocimiento de que va a ser adaptado. Cuando se necesita adaptar (añadir o modificar funcionalidad) el programa que se está ejecutando (porque surgen nuevos requerimientos o en respuesta a cambios en el entorno), sin detener la ejecución, se procesa el programa en ejecución (no el programa ejecutable, sino el que está en memoria) junto a los aspectos que van a adaptarlo, y por medio del tejedor se modifica el programa en memoria añadiéndole la funcionalidad de los aspectos, dando lugar a una nueva versión del programa que continúa la ejecución. El programa ejecutable inicial no ha sido modificado con lo que se puede utilizar de nuevo sin las modificaciones realizadas en memoria.

En sistemas que usan tejido dinámico de aspectos, la funcionalidad básica permanece separada de los aspectos en todo el ciclo de vida del *software*, incluso en la ejecución del sistema. El código resultante es más adaptable y reutilizable, y los aspectos y la funcionalidad básica pueden evolucionar de forma independiente [Pinto02].

Con el fin de superar las limitaciones del tejido estático han surgido diferentes sistemas que ofrecen tejido dinámico. Estos sistemas ofrecen al programador la posibilidad de modificar de forma dinámica el código del aspecto asignado a puntos de enlace de la aplicación de forma similar a los sistemas reflectivos en tiempo real basado en protocolos de meta objetos (*meta object protocols* o *MOPs*) [Maes87] [Kiczales91] [Sullivan01] [Baker02].

Muchas de las herramientas que afirman ofrecer un tejido dinámico de aspectos realmente ofrecen un híbrido entre el tejido estático y el dinámico, pues los aspectos deben conocerse y definirse en el momento del diseño e implementación, para posteriormente, en ejecución, poder instanciarlos. Aunque esta solución aporta alguna ventaja respecto al tejido estático hay casos en que no es suficiente, por ejemplo cuando una vez que la aplicación está funcionando surge un requerimiento nuevo, que no se tuvo en cuenta en el diseño.

Al igual que ocurre con las herramientas de tejido estático, la mayoría de las herramientas que ofrecen tejido dinámico son dependientes del lenguaje como se verá más adelante, si bien actualmente existen trabajos que tratan de eliminar esta restricción [Vinuesa07].

SISTEMAS DINÁMICOS

De cara a mostrar cómo se implementa la *POA* dinámica y el soporte ofrecido por las herramientas que permiten trabajar con ella, presentamos a continuación una serie de sistemas que teóricamente soportan la programación orientada a aspectos dinámica a partir de la clasificación realizada en [Vinuesa03].

PROSE

El sistema *PROSE* (*PROgramable extenSions of sErvices*) [Popovici01] [Popovici02], ofrece *POA* de forma dinámica, permitiendo adaptar en tiempo de ejecución una aplicación sin necesidad de haber definido nada en el momento del diseño. Su plataforma está implementada en *Java*, siendo éste el lenguaje de codificación de los aspectos y provee al usuario con un subconjunto de las características esenciales de la *POA* [Popovici02].

Para implementar este sistema se ha utilizado la interfaz de depuración de la máquina virtual de *Java*, *JVM Debugger Interface (JVMDI)* y se ha creado un *plug-in* (añadido) para la *JVM* de tal forma que soporte el concepto de aspecto de forma directa. Este *plug-in* recibe el nombre de interfaz de aspectos de la *JVM* (*JVM Aspect Interface, JVMAI*) que es el que permite al usuario el tejido dinámico de aspectos. Una limitación que presenta [Popovici02] es la imposibilidad de añadir nuevos miembros a una clase en el código original, lo que hace que no posea reflexión estructural. Al tener que utilizar *Java* como lenguaje base y lenguaje de los aspectos se tiene dependencia del lenguaje, lo que imposibilita su uso para programas no escritos en *Java*.

Con posterioridad [Popovici03] y con la intención de ofrecer mejor rendimiento en ejecución se ha modificado el compilador "justo a tiempo" (*Just In Time* o *JIT*) de la máquina virtual de investigación *Jikes* de *IBM* (*IBM Jikes Research Virtual Machine*) [IBM06], haciendo que ésta *JVM* soporte de forma directa la POA, y mediante una *API*, *Application Program Interface* (interfaz de programación de aplicaciones), se ofrecen puntos de enlace a la capa superior. Sobre este motor de ejecución se ha adaptado *PROSE* obteniendo buenos resultados en cuanto a tiempos de ejecución, pero sigue teniendo las mismas limitaciones que la versión anterior (dependencia del lenguaje y puntos de enlace limitados) y le añaden una nueva: la necesidad de utilizar una *JVM* modificada, no una estándar. Los autores del modelo admiten que el conjunto de puntos de enlace que soporta tiene limitaciones y por ejemplo no permite añadir *advice* (código) a niveles de objeto.

CLAW

El sistema *CLAW* (*Cross-Language Load-Time Aspect Weaving*, tejido de aspectos en tiempo de carga de lenguajes cruzados) [Lam02], inicialmente llamado *RAW* (*Runtime Aspect Weaver*, tejedor de aspectos en tiempo de ejecución) está implementado sobre la plataforma *.NET*. El trabajo lo realiza a nivel de código intermedio (*CIL*) obteniendo con ello un gran beneficio: la independencia del lenguaje.

En la plataforma *.NET* cualquier programa escrito en cualquier lenguaje de programación es traducido al código intermedio (*CIL*) que es el que se ejecuta por parte del *CLR* (*Commom Language Runtime*). Al introducir los aspectos a este nivel se consigue que cualquier programa escrito en cualquier lenguaje pueda ser adaptado por aspectos escritos en cualquier otro lenguaje, ya que al final ambos son compilados a *CIL* y es en este nivel donde son tejidos.

Como se puede ver por el nombre inicial (*RAW*) y por el definitivo (*CLAW*) en un principio se argumentaba que este sistema ofrecía tejido en tiempo de ejecución para, posteriormente afirmar que el tejido es en tiempo de carga. En realidad en este sistema, los aspectos tienen que estar definidos previamente (en tiempo de diseño), aunque se tejen en tiempo de ejecución, lo que implica que el sistema no puede adaptarse ante nuevos requerimientos no previstos en el diseño.

La forma de trabajar de este sistema es utilizar la *API* de *profiling* (*perfilar*) que provee *.NET*. En concreto los dos siguientes interfaces: *IcorProfilerCallback* e *IcorProfilerCallbackInfo*. Estos *interfaces* funcionan mediante un sistema basado en eventos. Cuando el sistema de aspectos empieza a ejecutarse informa al motor de ejecución de los eventos que quiere monitorizar. Los eventos más importantes a monitorizar serian la carga de módulos, carga de "*assemblies*" (ficheros ensamblados ejecutables) o la compilación justo a tiempo (*JIT*). En tiempo de ejecución el motor de ejecución llamaría al sistema cuando se produjesen esos eventos, y éste se encargaría (mediante reflectividad) de examinar el *CIL* existente, adaptarlo según sea necesario, y escribir el nuevo *CIL* adaptado en memoria para que el compilador *JIT* se encargue de compilarlo y ejecutarlo.

El principal inconveniente de esta forma de trabajar es que el sistema tiene que

estar en modo *profiling*, lo que implica una penalización en el rendimiento. Para obtener la reflectividad necesaria el autor ha utilizado los interfaces de metadatos no gestionado (*Unmanaged Metadata Interfaces*) los cuales son una conjunto de interfaces *COM* que están accesibles desde fuera del entorno *.NET* lo que hace que no sea portable al ser tecnología propia de *Microsoft*.

LOOM.NET

Este sistema desarrollado en el Hasso Plattner Institute ofrece en principio un tejido estático de aspectos [Schult02] pero con posterioridad [Schult02b] y [Schult03] le ha sido añadido una parte de tejido dinámico. Este sistema utiliza ficheros *XML* para definir el aspecto y las reglas del tejido. En estas reglas se define que es lo que se quiere ampliar (clase, constructor, método y campo son las posibilidades). El tejedor crea unas clases delegadas (*proxy classes*) para sustituir a las clases originales.

Aunque el sistema se ha implementado para *C#* los autores argumentan que es extensible para cualquier lenguaje dentro de la plataforma *.NET*. Se utilizan "*custom attributes*" (atributos personales) de *C#* para definir los puntos de enlace, y mediante introspección y reflectividad son evaluados en tiempo de ejecución.

El sistema presenta una limitación muy importante y es que sólo se permite la existencia de un aspecto por clase o método, impidiendo así que un elemento pueda ser ampliado con diferentes aspectos a la vez. Otra limitación es que sólo se puede añadir código en las clases, constructores, métodos y campos, pero sin control de si queremos hacerlo en la ejecución de un método o en su invocación, o de si queremos añadir antes o después de la funcionalidad; es decir, el conjunto de puntos de enlace es muy limitado. Además es necesario que al programar el aspecto se tenga conocimiento de los nombres de los campos, métodos, etc. que se quieren adaptar, y si, por ejemplo, se quisiese hacer un aspecto traza, que mostrase por pantalla un mensaje al entrar en un método implicaría conocer el nombre de todos los métodos que se quieren incluir en esta traza.

La parte dinámica consiste únicamente en la posibilidad de instanciar los aspectos en tiempo de ejecución, pero no se permite la creación de aspectos no contemplados en tiempo de diseño, es pues una característica limitada. Al igual que ocurría en *CLAW* se hace uso de los interfaces de metadatos no gestionados (*Unmanaged Metadata Interfaces*) con la misma pérdida de portabilidad mencionada antes.

AORTA

El sistema *AORTA* (*Aspect Oriented RunTime Architecture*, arquitectura orientada a aspectos en tiempo de ejecución) [AORTA06] pretende ofrecer un entorno con programación orientada a aspectos en tiempo de ejecución y está desarrollado por el mismo grupo que ha desarrollado *CAESAR* [Mezini03] (que es una extensión a *Java* para ofrecer orientación a aspectos en el mismo sentido que *AspectJ*).

El principal objetivo de este proyecto es lograr una implementación eficiente y flexible de *joinpoints* dinámicos, de manera que se integre soporte para los mismos dentro de los entornos de ejecución. *AORTA* intenta establecer una arquitectura general para entornos de ejecución que usen aspectos.

Para lograr sus objetivos, el proyecto se ha dividido en varias ramas. Una de ellas intenta implementar máquinas virtuales con mecanismos integrados para soportar aspectos dinámicos. Otra se ocupa de medir el rendimiento del *software* orientado a aspectos en general. Como resultado del trabajo realizado por la primera de estas ramas, *AORTA* consta de los siguientes subproyectos:

- ***Axon***: un *plugin* de *JVM* para soportar aspectos dinámicos, basado en la infraestructura de depuración de la máquina (*JVM debugger infrastructure*)
- ***RuByCom***: Otra extensión de la *JVM* para soportar aspectos dinámicos, pero basada en la tecnología *HotSwap* [HotSwap06]. Esta extensión de la *JVM* de *Sun* 1.4.2 y superiores permite el tejido dinámico de aspectos mediante la modificación de los *bytecodes* de los métodos. Los métodos modificados son reinstalados en la máquina virtual usando las capacidades de la tecnología *HotSwap* mencionada. Esta tecnología permite crear un modelo para joinpoints más potente que el empleado en sistemas anteriores (como *Axon*) y también permite superar las limitaciones existentes si se usa una técnica basada en la intercepción de eventos del depurador.
- ***Steamloom***: Es considerada por sus autores como la primera *JVM* capaz de soportar de manera nativa aspectos dinámicos. También es el objetivo principal del proyecto *AORTA* que estamos describiendo. El soporte para aspectos dinámicos esta integrado con la máquina virtual, que ofrece esa funcionalidad a la que se puede acceder a través de un *API* específico. Al igual que el sistema anterior, se basa en la capacidad de modificar y reinstalar *bytecodes* de métodos, pero tiene como ventaja el poseer un soporte integrado para diversas tareas relacionadas con el uso de aspectos. Esta implementación está basada en la máquina virtual *Jikes* de *IBM* [IBM06], a la que se ha incorporado un *toolkit* para manipular *bytecodes* llamado *BAT*, que permite modificar el *bytecode* y acceder a él a voluntad. Además, la recompilación de los métodos ocurre con todas las optimizaciones poseídas por el sistema.

EAOP

El nombre de este sistema viene de *Event-Based AOP* (*POA* basada en eventos) [Douence01]. La propuesta que se hace [Douence02] es un modelo teórico mediante el que se puede conseguir *EAOP* y una herramienta que lo implementa en *Java*. La *EAOP* tiene los siguientes conceptos:

- **Eventos**: que se generan durante la ejecución del programa con el propósito de designar "puntos de interés" ("*points of interest*") o puntos de ejecución.
- ***Crosscuts***: que son secuencias de eventos y relaciones entre ellos.
- **Monitorización** en ejecución de los eventos.

Las principales características que presenta son:

- Posibilidad de combinar aspectos de forma explícita.
- Posibilidad de crear aspectos que adapten a otros aspectos.
- Gestión dinámica de aspectos.

Inicialmente el modelo pretende ser un banco de pruebas para definir lenguajes orientados a aspectos (cualquier lenguaje). Se basa en la monitorización de eventos de ejecución. Tiene dos lenguajes, el de aspectos, y otro para definir los puntos de ejecución (que son como los *pointcuts* de *AspectJ*). En principio se puede elegir cualquier lenguaje para que actúe como lenguaje de aspectos, pero una vez seleccionado uno todos los

aspectos tienen que estar en ese lenguaje.

El sistema permite instanciación y composición dinámica de aspectos, pero los aspectos se deben definir en diseño. El sistema, mediante un preprocesador, procesa el código fuente del programa base y modifica el código mediante *method wrapping* insertando llamadas a sus librerías que son las que permiten gestionar los aspectos.

Los principales inconvenientes de este sistema son la dependencia del lenguaje, puesto que pese a que el sistema pretende ser independiente, en cuanto se ha elegido un lenguaje ya es necesario seguir utilizándolo, la necesidad de disponer del código fuente para poder adaptar a un programa, y la necesidad de haber definido los aspectos en el diseño, aunque sean instanciados posteriormente. Otra limitación muy importante es el muy restringido conjunto de puntos de enlace que soporta (llamada y retorno de un método o de un constructor), lo que limita mucho los aspectos a implementar. Tampoco permite concurrencia en la ejecución de aspectos.

DAOF

El nombre de este sistema viene de *Dynamic Aspect Oriented middleware Framework* y fue presentado en [Pinto01] y [Pinto02]. El sistema utiliza el lenguaje *Java* como lenguaje base y de aspectos. Este sistema es de dominio específico, estando orientado a entornos virtuales colaborativos (*Collaborative Virtual Enviroments*).

El sistema ofrece un tejido dinámico de aspectos, pero los aspectos deben ser definidos previamente en el diseño. Durante la fase de diseño se especifica la arquitectura de la aplicación mediante un lenguaje definido en *XML*. La primera parte de este lenguaje se utiliza para describir los componentes y el nombre del rol que cada componente juega dentro de la aplicación. A continuación el lenguaje se utiliza para describir los interfaces de los aspectos; Finalmente se describen las conexiones entre los componentes y los aspectos.

Este sistema presenta una serie de inconvenientes como son la dependencia del lenguaje (*Java*) y la necesidad de definir los aspectos en la fase de diseño de la aplicación, por lo que su utilidad está restringida. Otra limitación es la necesidad de disponer del código de los componentes, para poder modificarlo, ya que es necesario que extiendan una clase *component*. Además, al ser de dominio específico, no se puede emplear para implementar cualquier tipo de aspecto.

MICRODYNER (µDyner)

Este sistema fue presentado en [Segura-Devillechaise03] y [Segura-Devillechaise03b]. Surge en respuesta a la necesidad que encuentran los autores de poder implementar distintas políticas de "precarga en memoria" *cache prefetching* en los servidores *Web* de forma dinámica. Los autores han identificado la *POA* como una herramienta adecuada para poder implementar diversas políticas que puedan ir cambiando para adaptarse a requerimientos que varían en el tiempo.

La razón para crear un nuevo sistema es que la gran mayoría de los sistemas no ofrecen *POA* dinámica sino estática, y un servidor *Web* no puede detener su ejecución para responder a cambios en el entorno con lo que no es válido su uso. Otra razón es que los sistemas que ofrecen *POA* de forma dinámica no lo hacen para el lenguaje *C*, que es en el que están escritos la mayoría de los servidores *Web*, con lo que tampoco son aplicables.

La forma de trabajar con este sistema implica que en el programa fuente (el que va a ser adaptado) es necesario indicar qué partes pueden ser adaptadas (es decir, en el

diseño e implementación del programa a ser adaptado hay que tener en cuenta la posibilidad de una futura adaptación). Sólo las funciones y variables globales pueden ser candidatas a ser adaptadas. El programa pasa por un preprocesador antes de pasar por el compilador. Los aspectos se escriben en un lenguaje específico del sistema definido a tal efecto, son compilados mediante *µDyner* y posteriormente tejidos en tiempo de ejecución. El tejido se realiza directamente sobre la imagen ejecutable del programa base generando código ejecutable. El sistema permite realizar el tejido o el destejido (eliminación de un aspecto) en muy poco tiempo, por lo que el funcionamiento del servidor *Web* no se ve interrumpido.

En resumen este sistema permite crear aspectos únicamente para programas escritos en el lenguaje *C*, que pueden ser tejidos y destejidos en tiempo de ejecución, presentando los inconvenientes de tener que definir en el programa a adaptar qué es lo que puede ser adaptado y qué no, utilizar un lenguaje específico y sólo poder adaptar funciones y variables globales. Además al realizar el tejido o destejido directamente sobre el código ejecutable se incurre en una dependencia de la plataforma.

3.4.4.5 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Un sistema de *POA* estática, al igual que los sistemas vistos en la sección anterior, tiene como principal inconveniente la incapacidad de poder amoldarse a situaciones no previstas en fase de desarrollo. Las operaciones de depuración son otra de las desventajas que poseen la mayoría de sistemas que soportan *POA*. A nivel sintáctico un programa que use *POA* correctamente está completamente modularizado, pero esto no ocurre en tiempo de ejecución. Si no hay una especificación clara de las operaciones a realizar, cuando se une el código funcional con las incumbencias encapsuladas en aspectos puede haber un comportamiento impredecible a la hora de depurar. Se han propuesto formas alternativas de conseguir la separación de código, como los tipos parciales de *C#* [Bradley06], pero aún existen problemas que deben ser resueltos.

Otro problema presente en *POA* puede ocurrir con la definición de expresiones regulares que determinan los puntos de corte. Si bien estas expresiones determinarán, en el momento de definirse, un conjunto de métodos determinado al que el programador quiere unir un determinado aspecto, es posible que modificaciones posteriores en el código del programa hagan que nuevos métodos entren dentro de los criterios establecidos por la expresión de un punto de corte, ejecutando el aspecto sobre un método que inicialmente no estaba previsto. No es posible exigir a los desarrolladores que tengan en cuenta todas las posibles expresiones que se empleen para establecer puntos de unión en los métodos de una aplicación, ya que se incrementaría la dificultad de la elaboración del programa, por lo que encontrar una solución a este potencial problema no es una labor sencilla.

En lo relativo a la *POA* dinámica, la principal desventaja del tejido dinámico respecto al estático es el rendimiento en ejecución [Böllert99]. Los sistemas descritos también dejan entrever otro tipo de carencias existentes en este tipo de *POA*, como sistemas que no son realmente dinámicos (actúan en tiempo de carga, o requieren un conocimiento previo de los aspectos), sistemas que poseen limitaciones que les impiden implementar ciertas primitivas de reflexión (por ejemplo, imposibilidad de modificar las clases) o sistemas que usan como base máquinas no estándar o de investigación (por lo que sus posibilidades de aplicación fuera de los entornos en los que fueron creados son muy limitadas). En los dos primeros casos no podríamos lograr ningún avance real respecto a los sistemas estáticos en cuanto a su capacidad para ofrecer reflexión estructural. Por otra parte, el bajo rendimiento y el carácter experimental de la mayoría de estos sistemas limitan su aplicación como medios para obtener una plataforma eficiente que tenga capacidades de reflexión estructural dinámica y que pueda lograr un

uso extendido en el mercado.

Teniendo en cuenta el nivel de reflexión ofrecido, muchos de los sistemas estudiados presentan una mayor cantidad de información a reflejar que los que estudiaremos en el siguiente punto. La reflexión del lenguaje (o modificación de éste) presente en algunos de los casos estudiados se ofrece comúnmente en sistemas reflectivos estáticos, mientras que en los dinámicos su implementación no es frecuente.

Por otra parte, al efectuar la compilación previa del código fuente antes de pasar a la ejecución del mismo, se pueden utilizar lenguajes que comprueben sus tipos en tiempo de compilación [Cardelli97], lo que permite eliminar errores en tiempo de ejecución que pueden cometerse con sistemas de naturaleza interpretada, como *Python*, que comprueba sus tipos en tiempo de ejecución [Rossum01].

3.4.5 Sistemas Dotados de Refl. Computacional

3.4.5.1 REFL. COMPUTACIONAL BASADA *MOPS*

Los sistemas reflectivos que tienen la capacidad de modificar parte de su semántica dinámicamente (en tiempo de ejecución) son comúnmente implementados utilizando el concepto de protocolo de metaobjeto *(Meta-Object Protoco*l o *MOP*). Estos *MOPs* son un protocolo o modo de acceso del sistema base al metasistema, permitiendo modificar parte de su propio comportamiento dinámicamente siguiendo las normas establecidas por el protocolo. Por tanto, las capacidades dinámicas del sistema estarán limitadas y determinadas por las posibilidades ofrecidas por dicho protocolo.

En este punto se analizarán un conjunto de sistemas que utilizan *MOPs* para modificar su semántica dinámicamente, para posteriormente hacer una síntesis global de sus aportaciones y carencias, basándose en la clasificación realizada en [Ortin01].

CLOSETTE

Closette [Kiczales91] es un subconjunto del lenguaje de programación *CLOS* [Steele90]. Fue creado para implementar un *MOP* para el lenguaje *CLOS*, permitiendo modificar dinámicamente aspectos semánticos y estructurales del lenguaje. La implementación se basa en desarrollar un intérprete de este subconjunto de *CLOS* sobre el propio lenguaje *CLOS*, capaz de ofrecer un protocolo de acceso a su metasistema.

Existen pues dos niveles computacionales: el nivel del intérprete de *CLOS* (metasistema), y el del intérprete de *Closette* (sistema base) desarrollado sobre el primero. El acceso del sistema base al metasistema se realiza mediante un sistema de macros; el código *Closette* se expande a código *CLOS* que al ser evaluado puede acceder a su metasistema. La interfaz de estas macros es la especificación del *MOP* del sistema.

El diseño de este *MOP* para el lenguaje *CLOS* se centra en el concepto de metaobjeto. Un metaobjeto es cualquier abstracción (estructural o computacional) del metasistema susceptible de ser modificada por su sistema base. Al igual que otros sistemas que emplean el concepto de metaobjetos, un metaobjeto en este caso no es necesariamente una representación de un objeto en su sistema base, sino que puede representar una clase o algo más abstracto, como la semántica de la invocación a un método. El modo en el que se puede llevar a cabo la modificación de los metaobjetos es

definido por el propio *MOP*.

La implementación del sistema fue llevada a cabo en tres capas, mostradas en la figura 3.19:

- La capa de macros; define la forma en la que se va a interpretar el subconjunto de *CLOS* definido (*Closette*), estableciéndolo mediante traducciones a código *CLOS*. En el caso de que no existiese un *MOP*, esta traducción sería la identidad: el código *Closette* se traduciría a código *CLOS* sin ningún cambio.
- La capa de unión. Esta capa consiste en un conjunto de funciones desarrolladas en *CLOS* que facilitan el acceso a los objetos del metasistema, para así facilitar la implementación de la traducción de las macros. Como ejemplo, podemos citar la función `find-class` que obtiene el metaobjeto representativo de una clase, cuyo identificador es pasado como parámetro.
- La capa de nivel inferior. Es la representación en *CLOS* (metasistema) de la estructura y comportamiento del sistema base. Todas aquellas partes del sistema que deseen ser reflectivas, deberán ser implementadas como metaobjetos.

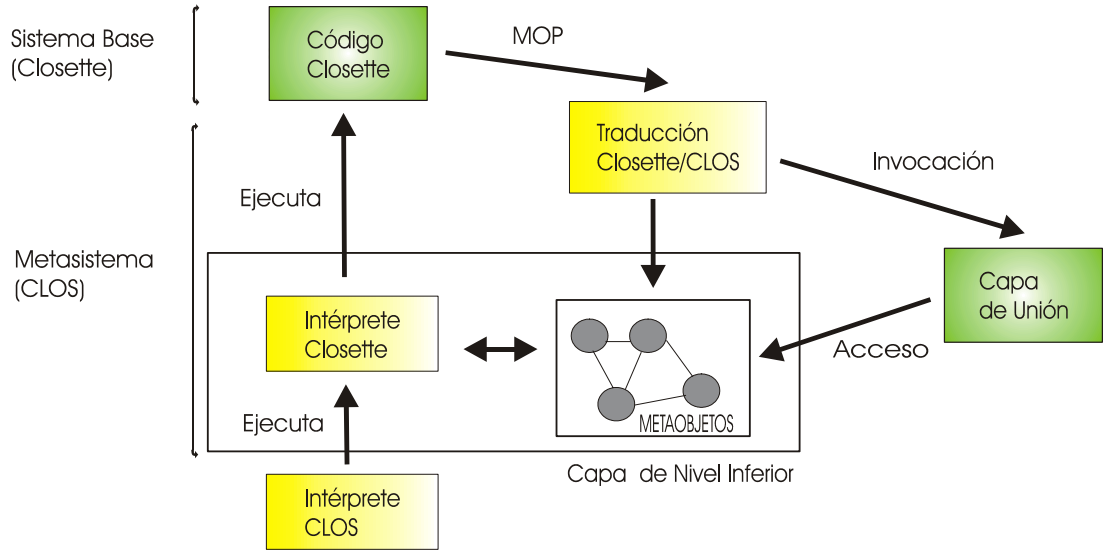

Figura 3.19: Arquitectura del *MOP* desarrollado para el lenguaje *CLOS*

METAXA

MetaXa (también llamado *MetaJava*) es un sistema basado en el lenguaje y plataforma *Java* a la que añade características reflectivas en tiempo de ejecución, permitiendo modificar parte de la semántica de la implementación de la máquina virtual [Golm97]. El diseño de la plataforma está orientado a dar soporte a la creación de aplicaciones flexibles que puedan adaptarse a requerimientos dinámicos como distribución, seguridad, persistencia, tolerancia a fallos o sincronización de tareas [Kleinöder96].

El modo en el que se deben desarrollar aplicaciones en *MetaXa* se basa en el concepto de metaprogramación *(metaprogramming)* [Maes87]: El código funcional se separa del código no funcional. La parte funcional de una aplicación se centra en el modelado del dominio de la aplicación (nivel base), mientras que el código no funcional

formaliza el modelado de determinados aspectos propios del código funcional (metasistema). *MetaXa* permite separar estos dos niveles de código fuente y establecer entre ellos un mecanismo de conexión causal en tiempo de ejecución.

El sistema de computación de *MetaXa* se apoya sobre la implementación de objetos funcionales y metaobjetos conectados a los primeros. A un metaobjeto se le puede conectar objetos, referencias y clases, aunque para hablar de forma general utilizaremos el término objeto para referirnos a estos tres tipos de elementos conectables. Cuando un metaobjeto está conectado a un objeto sobre el que sucede una acción, el sistema provoca un evento en el metaobjeto indicándole la operación solicitada en el sistema base. Por tanto, el código que se incluya dentro del metasistema para el evento asociado a una determinada acción permitirá modificar la semántica de lo que ocurre cuando se provoca el evento. La computación del sistema base se suspende de forma síncrona hasta que el metasistema finalice la interpretación del evento capturado, para evitar posibles errores derivados de esta situación.

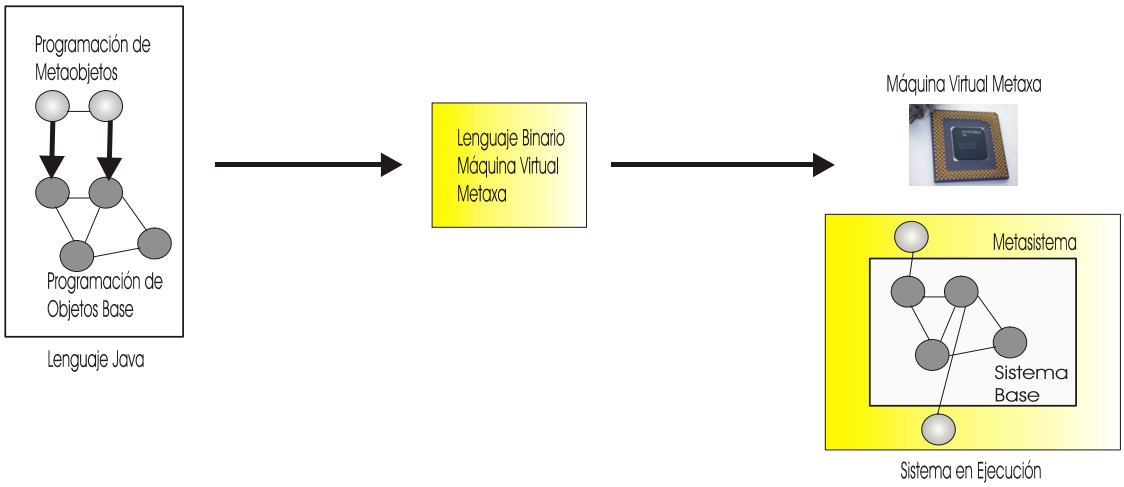

Figura 3.20: Fases en la creación de una aplicación en *MetaXa*

Por ejemplo, si un metaobjeto recibe el evento `method-enter`, la acción por omisión será ejecutar el método apropiado. Sin embargo, el metaobjeto podría implementar el método `eventMethodEnter` para modificar el modo en el que se interpreta la recepción de un mensaje, modificando de esta forma la semántica del sistema. En *MetaXa* ningún objeto está inicialmente conectado a un metaobjeto, con el objetivo de hacer el sistema lo más flexible posible. Las conexiones entre objetos y metaobjetos que respondan a las solicitudes hechas por el programador siempre se harán en tiempo de ejecución. La implementación de la plataforma reflectiva de *MetaXa* toma la máquina virtual de *Java* [Sun95] y añade sus nuevas características reflectivas mediante la implementación de métodos nativos residentes en una librería de enlace dinámico [Sun97c].

No obstante, la implementación de *Metaxa* no está exenta de problemas, derivados de las características del modelo de clases tradicional, donde el comportamiento y estructura de todo objeto debe estar reflejado en su clase. Uno de los inconvenientes posee proviene de la modificación de la estructura de una clase cualquiera. Si se modifica una clase, entonces todas las instancias de la misma deberán modificarse en consonancia, necesitando pues un mecanismo que haga de forma transparente dichos cambios a un coste razonable. Otro de los principales inconvenientes de su diseño es la modificación del comportamiento individual de un objeto, ya que si se desea modificar la semántica de una sola instancia de dicha clase, entonces la clase asociada a dicha instancia no cumpliría con la norma mencionada. La solución que *MetaXa* da a este problema es crear una nueva clase para el objeto, denominada clase

sombra *(Shadow Class),* y que contendría el reflejo de los cambios hechos para esa única instancia, de manera que no se pierda el concepto de tipo propio del paradigma orientado a objetos basado en clases y toda instancia tenga una clase que refleje su estructura. Las clases sombra tienen las siguientes características [Golm97c]:

- Una clase y su clase sombra asociada son iguales para el nivel base.
- La clase sombra difiere de la original en las modificaciones realizadas en el metasistema.
- Los métodos y atributos de clase (`static`) son compartidos por ambas clases.

Figura 3.21: Creación dinámica de una clase sombra en *MetaXa* para modificar el comportamiento de una instancia de una clase

No obstante este enfoque posee también una serie de problemas que hay que solventar para mantener coherente el sistema cuando se utilizan las clases sombra [Golm97c]:

- Consistencia de atributos. La modificación de los atributos de una clase ha de mantenerse coherente con su representación sombra.
- Identidad de clase. Hay que tener en cuenta que el tipo de una clase y el de su clase sombra han de ser el mismo aunque tengan distinta identidad, o sino romperíamos las reglas del modelo de objetos.
- Objetos de la clase. Cuando el sistema utilice el objeto representante de una clase (en *Java* una clase genera un objeto en tiempo de ejecución que permite operar con la estructura de la misma) entonces el objeto correspondiente a la clase original y el de su sombra han de tratarse paralelamente. Un ejemplo pueden ser las operaciones `monitorenter` y `monitorexit` de la máquina virtual [Venners98], que tratan los objetos clase como monitores de sincronización. En estos casos los monitores necesariamente tienen que se los mismos tanto para las clase sombra como para la real.
- Recolector de basura. Las clases sombra deberá limpiarse cuando el objeto no tenga asociado un metaobjeto.
- Consistencia de código. Tendrá que mantenerse cuando se produzca la creación de una clase sombra a la hora de estar ejecutándose un método de la clase original.
- Consumo de memoria. La duplicidad de una clase produce elevados consumos de memoria, por lo que son necesarias técnicas que minimicen estos problemas.
- Herencia. La creación de una clase sombra, cuando la clase inicial es derivada de otra, obliga a la producción de otra clase sombra de la clase base original. Este proceso ha de expandirse de forma recursiva, con el coste asociado.

La enorme complejidad surgida por el concepto de clase sombra, y todo lo que ello conlleva, han llevado a la cancelación de este proyecto. Los propios autores del sistema afirman que los lenguajes basados en prototipos como *Self* [Ungar87] o *Moostrap* solucionan estos problemas de una forma más coherente [Golm97c].

IGUANA

La mayoría de los sistemas reflectivos, adaptables en tiempo de ejecución y basados en *MOPs*, son desarrollados mediante la interpretación de código. La principal razón es que los intérpretes tienen que construir toda la información propia de la estructura y la semántica de la aplicación a la hora de ejecutarla, a diferencia de los sistemas basados en la ejecución de código nativo. Si un entorno reflectivo trata de modificar la estructura o comportamiento de un sistema, será más sencillo ofrecer esta información si la ejecución de la aplicación es interpretada, ya que está disponible cuando se ejecuta.

En cambio, en el caso de los compiladores, la información relativa al sistema es creada en tiempo de compilación (y generalmente almacenada en la tabla de símbolos [Cueva92b]), para llevar a cabo comprobaciones de tipos [Cueva95b] y generación de código [Aho90] pero, una vez compilada la aplicación, dicha información deja de existir. En el caso de un depurador *(debugge*r), parte de la información es mantenida en tiempo de ejecución para poder conocer el estado de computación (introspección) y permitir modificar su estructura (reflexión estructural dinámica), pero esto supone un coste adicional que supone un aumento de tamaño de la aplicación y una pérdida de rendimiento en ejecución.

El sistema *Iguana* ofrece reflexión computacional en tiempo de ejecución basada en un *MOP*, compilando código *C++* a la plataforma nativa destino [Gowing96]. En la generación de código, al contrario que un depurador, *Iguana* no genera información de toda la estructura y comportamiento del sistema. Por omisión compila el código origen *C++* a la plataforma destino sin ningún tipo de información dinámica. El programador ha de especificar qué parte del sistema desea que sea adaptable en tiempo de ejecución, de modo que el sistema generará el código oportuno para que sea reflectivo. *Iguana* define dos conceptos que especifican el grado de adaptabilidad de una aplicación:

- Categorías de cosificación *(Reification Categories*). Indican al compilador dónde debe producirse la cosificación del sistema. Son elementos susceptibles de ser adaptados en *Iguana,* como clases, métodos, objetos, creación y destrucción de objetos, invocación a métodos o recepción de mensajes, etc.
- Definición múltiple de *MOPs (Multiple fine-grained MOP*s). El programador ha de definir la forma en la que el sistema base va a acceder a su información dinámica, es decir se ha de especificar el *MOP* de acceso al metasistema.

La implementación de *Iguana* está basada en el desarrollo de un preprocesador que lee el código fuente *Iguana* (una extensión del *C++*) y traduce toda la información específica del *MOP* a código *C++*, con información dinámica adicional adaptable en tiempo de ejecución (el código *C++* no reflectivo no sufre proceso de traducción alguno). Una vez que la fase de preproceso haya sido completada, *Iguana* invocará a un compilador de *C++* para generar la aplicación final nativa, adaptable dinámicamente, pero sólo en aquellas partes que el programador haya especificado de antemano.

COGNAC

Cognac [Murata94] es un sistema orientado a objetos basado en clases, cuya intención es proporcionar un entorno de programación de sistemas operativos orientados a objetos como *Apertos* [Yokote92]. El lenguaje de programación de *Cognac* es similar a *Smalltalk-80* [Goldberg89], aunque para aumentar su eficiencia, se le ha añadido comprobación estática de tipos [Cardelli97]. Los principales objetivos del sistema son:

- Uniformidad y simplicidad: Para el programador sólo debe haber un tipo de objeto concurrente, sin diferenciar ejecución síncrona de asíncrona.
- Eficiencia: Necesaria para desarrollar un sistema operativo.
- Seguridad: Deberá tratarse de minimizar el número de errores en tiempo de ejecución, razón por la cual se han introducido tipos estáticos al lenguaje.
- Migración: En el sistema los objetos deberán poder moverse de una plataforma física a otra, para seleccionar el entorno de ejecución que más les convenga.
- Metaprogramación: El sistema podrán programarse separando las distintas incumbencias y aspectos de las aplicaciones, diferenciando el código funcional del no funcional.

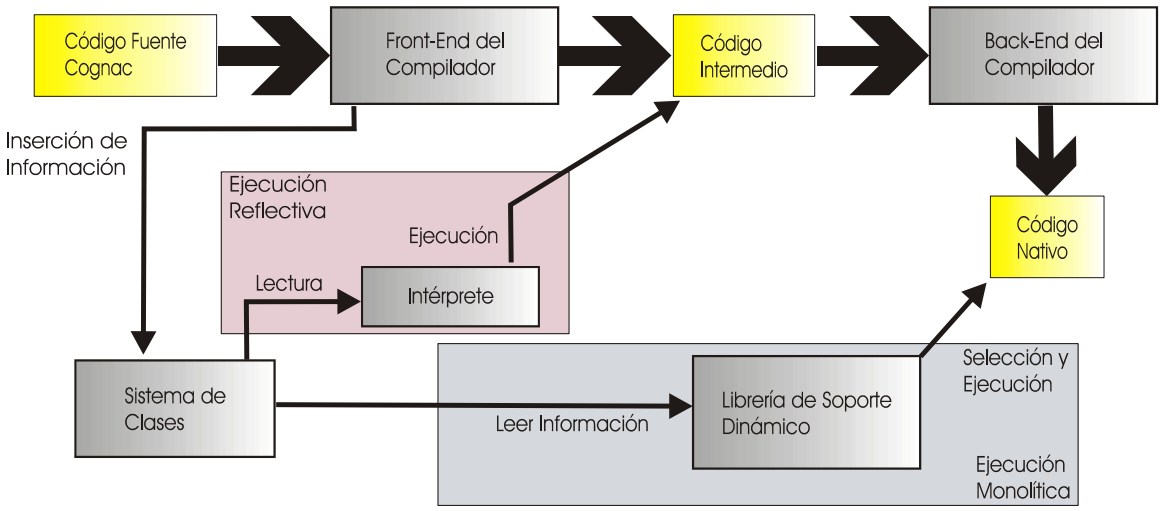

Figura 3.22: Arquitectura y ejecución de una aplicación en *Cognac*

La arquitectura del sistema está compuesta de cinco elementos:

- El *front-end* del compilador. El código fuente *Cognac* es traducido a un código intermedio independiente de la plataforma destino seleccionada. El compilador selecciona la información propia de las clases y la almacena, para su posterior uso, en el sistema de clases (último elemento).
- El *back-end* del compilador. Esta parte del compilador toma el código intermedio y lo traduce a código binario propio de la plataforma física utilizada. Crea un conjunto de funciones traducidas de cada rutina del lenguaje de alto nivel.
- Librería de soporte dinámico *(Runtime Support Library)*. Es el motor principal de ejecución. Envía una petición al sistema de clases para conocer el método apropiado del objeto implícito a invocar. Una vez identificado éste en el código nativo, carga la función apropiada y la ejecuta.

- Intérprete. Aplicación nativa capaz de ejecutar el código intermedio de la aplicación. Será utilizado cuando el sistema requiera reflejarse dinámicamente. La ejecución del código en este modo es menos eficiente que la ejecución nativa.
- Sistema de clases. Contiene la información dinámica relativa a las clases y métodos del sistema. El proceso de reflexión dinámica y los papeles de las distintas partes del sistema se muestran en la Figura 3.22. La ejecución del sistema es controlada por la librería de soporte dinámico, que lee la información relativa al mensaje solicitado, busca éste en el código nativo y lo ejecuta. Cuando se utiliza un metaobjeto dinámicamente, el motor de ejecución pasa a ser el intérprete, que ejecuta el código intermedio del mismo. El comportamiento del sistema es determinado por la interpretación del metaobjeto creado, dictando éste la nueva semántica del sistema.

GUANARÁ

Para el desarrollo de la librería *MOLDS* [Oliva98] de componentes de metasistema reusables, enfocados a la creación de aplicaciones de naturaleza distribuida, y que además ofrezcan características de persistencia, distribución y replicación indiferentemente de la aplicación desarrollada (lo que constituiría una verdadera separación de incumbencias), se desarrolló la plataforma computacionalmente reflectiva *Guanará* [Oliva98b]. *Guanará* es una ampliación de la implementación de la máquina virtual de *Java Kaffe OpenVM*, otorgándole la capacidad de ser reflectiva computacionalmente en tiempo de ejecución mediante un *MOP* [Oliva98c].

El mecanismo de reflexión ofrecido al programador está centrado en el concepto de metaobjeto. Cuando se enlaza un metaobjeto a una operación del sistema, la semántica de dicha operación es derogada por la evaluación del metaobjeto. El modo en el que sea desarrollado este metaobjeto definirá dinámicamente el nuevo comportamiento de la operación modificada.

El concepto de metaobjeto es utilizado por multitud de *MOPs* y la combinación de éstos se suele llevar a cabo mediante el patrón de diseño "Cadena de Responsabilidad" *(Chain of Responsibility)* [GOF94]. En este patrón, cada metaobjeto es responsable de invocar al siguiente metaobjeto enlazado con su misma operación, y devolver su resultado. Este esquema de funcionamiento es poco flexible, ya que todo metaobjeto necesitaría modificar su comportamiento en función del resto (Figura 3.23).

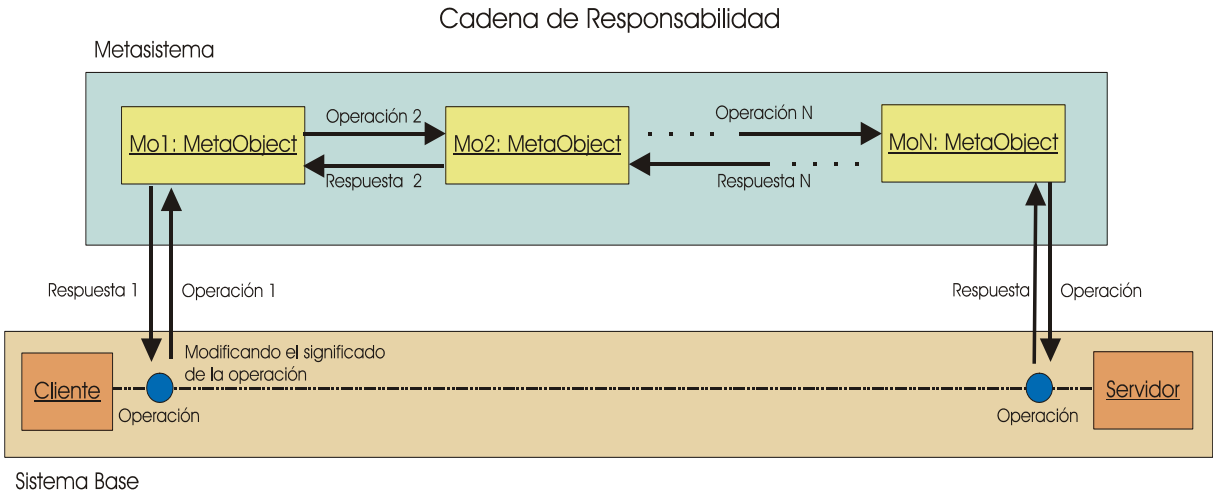

Figura 3.23: Utilización del patrón *Chain of Responsibility* para asociar múltiples metaobjetos

Guanará aborda este problema con el uso del patrón "Composición" *(Composit*e) [GOF94], permitiendo establecer combinaciones de comportamiento más independientes y flexibles [Oliva99]. Cada operación puede tener enlazado un único metaobjeto primario, denominado compositor *(compose*r). Éste, haciendo uso del patrón "Composición", podrá acceder a una colección de metaobjetos mediante una estructura de grafo. Como se muestra en la Figura 3.24, cada uno de los elementos de un compositor puede ser un metaobjeto u otro compositor.

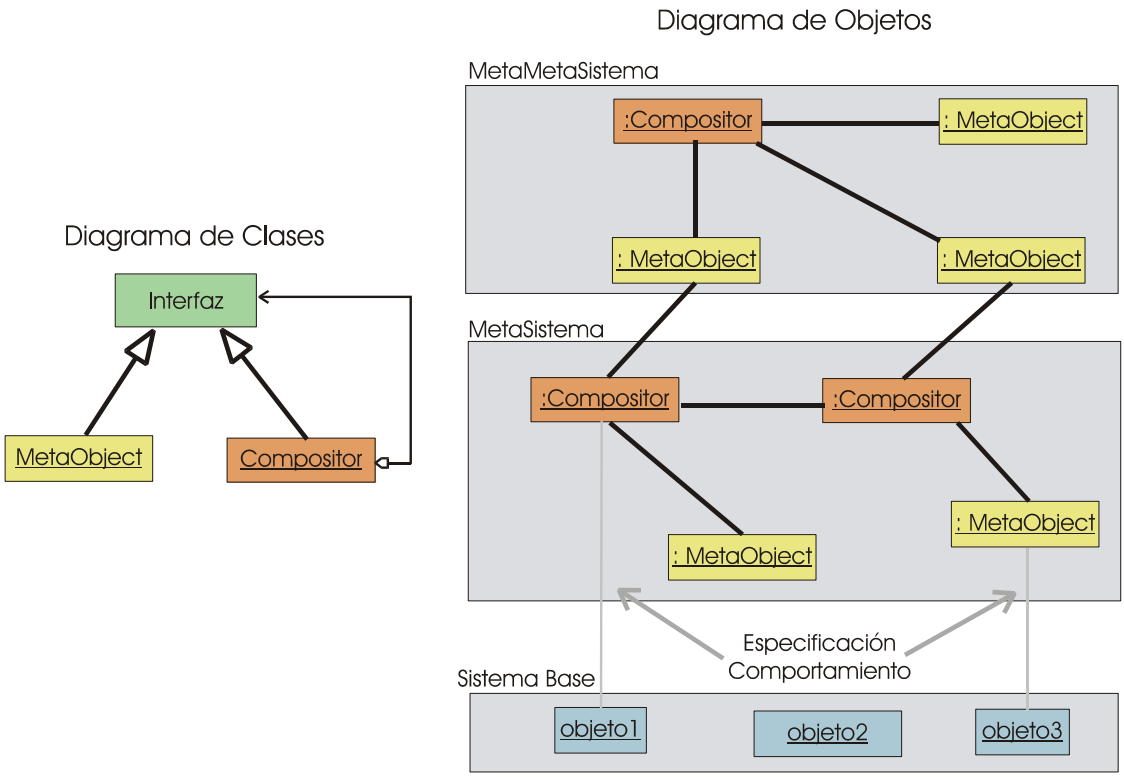

Figura 3.24: Utilización del patrón `Composite` para asociar múltiples metaobjetos

El modo en el que cada compositor establece el nuevo comportamiento de su operación asociada, en función de los metaobjetos utilizados, viene definido por una configuración adicional, denominada metaconfiguración. De esta forma, se separa la estructura de los metaobjetos de su secuencia de evaluación, haciendo el sistema más flexible y reutilizable. La parte realmente novedosa de este sistema radica en la forma en la que se pueden combinar múltiples comportamientos a una determinada operación reflectiva, así como el establecimiento de metacomportamientos de un metaobjeto (metametaobjetos), estableciendo en conjunto un mecanismo escalable y flexible, basado en el patrón de diseño *Composit*e.

DALANG

Dalang [Welch98] es una extensión reflectiva del *API* de *Java* [Kramer96], que añade reflexión computacional para modificar únicamente el paso de mensajes del sistema de un modo restringido. Este sistema implementa dos mecanismos de reflexión: en tiempo de compilación (estática) y en tiempo de ejecución (dinámica).

El conjunto de clases utilizadas en el esquema estático se muestra en la Figura

3.25. Dada una clase *C* cuyo paso de mensajes deseemos modificar, *Dalang* sustituye dicha clase por otra nueva con la misma interfaz y nombre (renombrando la original), siendo esto posible gracias a la introspección de la plataforma *Java* [Sun97d]. La nueva clase posee un objeto de la clase original en la que delegará la ejecución de todos sus métodos. Sin embargo, poseerá la capacidad de ejecutar, previa y posteriormente a la invocación del método, código adicional que pueda efectuar transformaciones de comportamiento (en la Figura 3.25, este código reside en la implementación de los métodos `beforeMethod` y `afterMethod`).

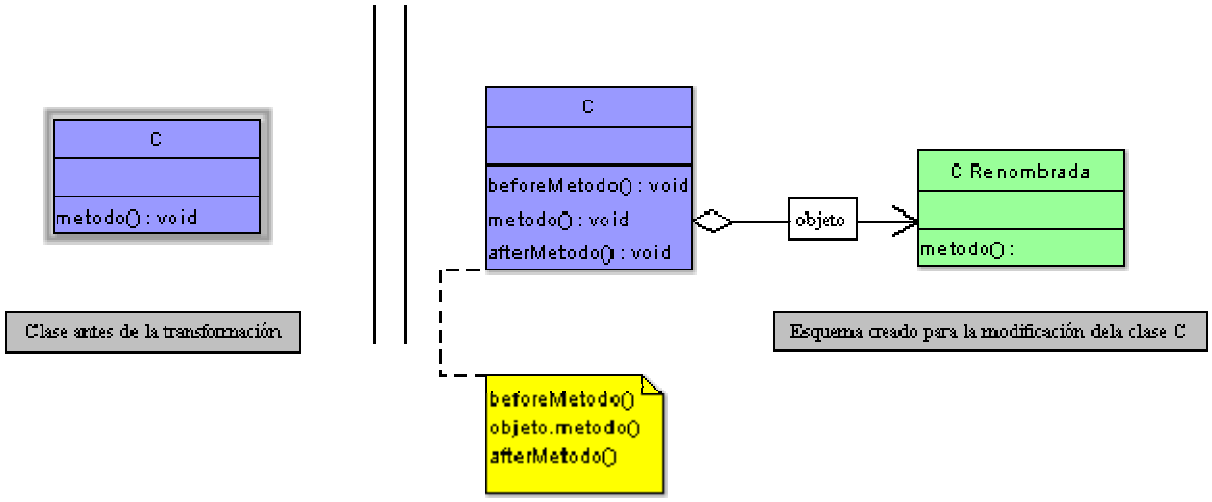

Figura 3.25: Transformación de una clase en *Dalang* para obtener reflexión

La reflexión dinámica del sistema se obtiene añadiendo al esquema anterior la implementación de un cargador de código capaz de realizar la carga dinámica de las nuevas clases en el sistema. La plataforma *Java* ofrece fácilmente esta posibilidad mediante la implementación de una clase derivada de *ClassLoader* [Gosling96]. Esta arquitectura posee un conjunto de inconvenientes:

- Transparencia. Se permite modificar el paso de mensajes de un conjunto de objetos (las instancias de la clase renombrada), pero no es posible modificar la semántica del paso de mensajes para todo el sistema.
- Grado de reflexión. La modificación de la semántica se reduce al paso de mensajes y se realiza en un grado bastante reducido: Ejecución anterior y posterior de código adicional.
- Eficiencia. La creación dinámica de clases requiere una compilación al código nativo de la plataforma, con la consecuente ralentización en tiempo de ejecución.

Su principal ventaja es que no modifica la máquina virtual de *Java* ni el código existente en su plataforma. De esta forma, el sistema no pierde la portabilidad del código *Java* y es compatible con cualquier aplicación desarrollada para esta plataforma virtual.

NEOCLASSTALK

Tras los estudios de reflexión estructural llevados a cabo con el sistema *ObjVlisp* analizado anteriormente, la arquitectura evolucionó hacia una ampliación de *Smalltalk*

[Goldberg83] denominada *Classtalk* [Mulet94]. El propósito de este sistema era utilizar esta plataforma como medio de estudio experimental para el desarrollo de aplicaciones con reflexión estructural. El desarrollo de un *MOP* que permitiese modificar el comportamiento de las instancias de una clase hizo que el sistema se renombrase a *NeoClasstalk* [Rivard96].

La aproximación que se utilizó para añadir reflexión computacional a *NeoClasstalk* fue la utilización del concepto de metaclase[7] propio del lenguaje *Smalltalk*. Una metaclase define la forma en la que van a comportarse (semántica computacional) sus instancias (que son sus clases asociadas, al ser una metaclase). Sobre este sistema se desarrollaron metaclases genéricas para poder ampliar las características del lenguaje [Ledoux96], tales como la definición de métodos con pre y poscondiciones, clases de las que no se pueda heredar o la creación de clases que sólo puedan tener una única instancia (el patrón de diseño *Singleton* [GOF94]). El modo en el que se modifica el comportamiento se centra en la modificación dinámica de la metaclase de una clase. Como se muestra en la Figura 3.26, existe una metaclase por omisión denominada `StandardClass`. Esta metaclase define el comportamiento general de una clase en el lenguaje de programación *Smalltalk*. El nuevo comportamiento deseado deberá implementarse en una nueva metaclase, derivada de `StandardClass`.

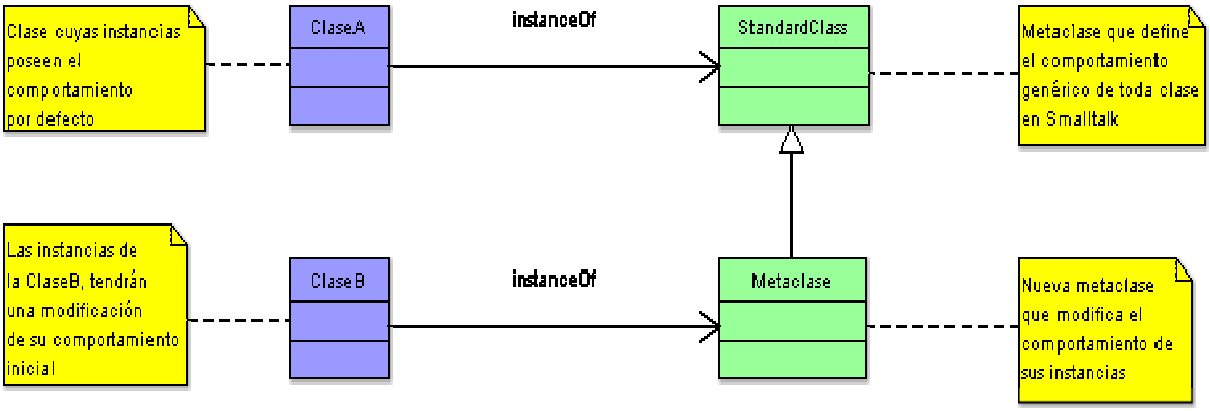

Figura 3.26: *MOP* de *NeoClasstalk* utilizando metaclases

En *NeoClasstalk*, los objetos pueden cambiar de clase dinámicamente, modificando su asociación `instanceof`. Si modificamos esta referencia en el caso de una clase, se modificará su metaclase y, por tanto, también su comportamiento. Una de las utilizaciones prácticas de este sistema fue el desarrollo de *OpenJava* [Ledoux99], un *ORB* capaz de adaptarse dinámicamente a los requisitos del programador *CORBA* [OMG98]. Mediante la modificación del comportamiento basado en metaclases se añade una mayor flexibilidad al *middleware CORBA* consiguiendo las siguientes funcionalidades:

- Modificar dinámicamente el protocolo de comunicaciones. La utilización de objetos delegados *(proxy)* permite modificar el comportamiento del paso de mensajes, seleccionando dinámicamente el protocolo deseado.
- Migración de objetos servidores dinámicamente.
- Replicación de objetos servidores.

7 Al igual que en los lenguajes orientados a objetos basados en clases, un objeto es una instancia de una clase. El concepto de metaclase define una clase como instancia de una metaclase, que define el comportamiento de todas sus clases asociadas.

- Implementación de un sistema dinámico de caché.
- Gestión dinámica de tipos. Accediendo dinámicamente a las especificaciones de los interfaces de los objetos servidores (archivos *IDL*), se pueden implementar comprobaciones de tipo en tiempo de ejecución.

El desarrollo del sistema fue llevado a cabo siguiendo la separación de incumbencias: la parte de una aplicación que modele el dominio del problema se separa del resto de código, que de esta forma podrá ser reutilizado para otras aplicaciones, y modele un aspecto global a varios sistemas.

MOOSTRAP

Moostrap [Mulet93] es un lenguaje orientado a objetos reflectivo basado en prototipos, implementado como un intérprete desarrollado en el lenguaje *Scheme* [Abelson00]. Como la mayoría de los lenguajes basados en prototipos, define un número reducido de primitivas computacionales que se van extendiendo mediante la utilización de sus características de reflexión estructural, para ofrecer un mayor nivel de abstracción en la programación de aplicaciones. En el próximo capítulo se hará una descripción detallada de las características de este tipo de lenguajes basados en prototipos.

Mediante sus capacidades estructuralmente reflectivas, este sistema extiende las primitivas iniciales para ofrecer un mayor nivel de abstracción al programador de aplicaciones. Un ejemplo de esto es la definición del mecanismo de herencia apoyándose en su reflexión estructural dinámica [Mulet93]: si un objeto recibe un mensaje y no tiene un miembro con dicho nombre, se obtiene su miembro `parent` y se le envía dicho mensaje a este objeto, continuando este proceso de un modo recursivo.

Además de reflexión estructural, *Moostrap* define un *MOP* para modificar el comportamiento de selección y ejecución de un método de un objeto ante la recepción de un mensaje. La semántica de la derogación de estas dos operaciones viene definida por el concepto de metaobjeto [Kiczales91], utilizado en la mayor parte de los *MOPs*. El paso de mensajes puede modificarse en dos fases: primero, ejecutándose el metaobjeto asociado a la selección del miembro y, posteriormente, interpretando el comportamiento definido por el metaobjeto que derogue la ejecución del método. Utilizando *Moostrap*, se trató de definir una metodología para crear metacomportamientos mediante la programación de metaobjetos en *Moostrap* [Mulet95].

APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

En la totalidad de los *MOPs* estudiados se permite la modificación dinámica de una parte del comportamiento del mismo y, en algunos casos (*Dalang*), dichas modificaciones se pueden realizar en tiempo de compilación para lograr una mayor eficiencia en tiempo de ejecución. El concepto de *MOP* establece un modo de acceso del sistema base al metasistema, por lo que determina el comportamiento que puede ser modificado. Al estar establecido previamente a la ejecución de la aplicación, el *MOP* será una restricción a priori de la semántica que podrá ser modificada dinámicamente, limitando pues lo que un sistema puede modificar en su propio comportamiento. No obstante, existen técnicas para tratar de resolver esta limitación de los *MOPs,* como por ejemplo ampliar éste cuando sea necesario [Golm98]. Por ejemplo, si un *MOP* no contempla la modificación de la semántica de la creación de objetos, entonces se ampliará para que sea adaptable. Sin embargo, teniendo en cuenta las funcionalidades ofrecidas a los programas por un *MOP*, éste deberá estar muy ligado al intérprete, por lo que la modificación del *MOP* supone

también la modificación del intérprete, dando lugar a distintas versiones del mismo y a la pérdida de la portabilidad del código existente para las versiones anteriores [Ortín2001].

Los sistemas basados en *MOPs* otorgan una flexibilidad dinámica de su comportamiento y carecen de la capacidad de modificar el lenguaje con el que son desarrolladas sus aplicaciones, sino que sólo modifican la semántica de éstos. No obstante, la mayor carencia de este tipo de sistemas reflectivos es su eficiencia en ejecución. La utilización de intérpretes es más común, por lo que las aplicaciones finales poseen tiempos de ejecución más elevados que si hubieren sido compiladas a código nativo. A raíz de analizar los sistemas estudiados se puede decir:

- La implementación de un *MOP* en un sistema interpretado (por ejemplo, *MetaXa*) es más sencilla y menos eficiente que el desarrollo de un traductor a un lenguaje compilable, como por ejemplo *Iguana*. En este caso, el sistema debe poseer información dinámica adicional que pueda accederse y modificarse en tiempo de ejecución. Un ejemplo es la posibilidad de conocer dinámicamente el tipo de un objeto en *C++* (*RTTI, RunTime Type Information*) [Stroustrup98]. Esta característica supone modificar la generación de código para que todo objeto posea la información propia de su tipo, información que generalmente en la ejecución de una aplicación nativa no necesita y por tanto no se genera.
- Puesto que los sistemas compilados poseen mayor eficiencia frente a los interpretados, que ofrecen una mayor sencillez a la hora de implementar sistemas flexibles, la unión de las dos técnicas de generación de aplicaciones puede dar lugar a un compromiso eficaz. En el caso de *Cognac*, todo el sistema se ejecuta en tiempo dinámico excepto aquella parte que se identifica como reflectiva, momento en el cual un intérprete ejecuta el código intermedio que define el nuevo comportamiento. Para el sistema *Iguana* todo el código es traducido sin información dinámica, salvo aquél que va a ser adaptado.
- El desarrollo de un *MOP* en dos niveles de interpretación (*Closette*) es más sencillo que si sólo elegimos uno (*MetaXa*). Si necesitamos modificar el *MOP* de *Closette*, deberemos hacerlo sobre el primer intérprete. En el caso de *MetaXa*, deberemos recodificar la máquina virtual, proceso más complejo.
- Actualmente no existe ningún lenguaje que ofrezca *MOPs* y se utilice en el desarrollo de aplicaciones comerciales.

3.4.5.2 REFLEXIÓN COMPUTACIONAL BASADA EN INTÉRPRETES METACIRCULARES

Anteriormente se ha mencionado que concepto de reflexión computacional utilizando la metáfora de una torre de intérpretes, de manera que cada nivel suponía un nivel de abstracción mayor. De esta forma, cuando una aplicación quiera acceder a su nivel inferior, podrá modificar su comportamiento. Si una aplicación necesita modificar la semántica de su semántica (el modo en el que se interpreta su comportamiento), deberá acceder en la torre a un nivel computacional dos unidades inferior, etc.

El proceso de acceder desde un nivel en la torre de intérpretes definida por Smith [Smith82] a niveles inferiores puede, teóricamente, extenderse hasta el infinito, ya que todo intérprete será ejecutado o animado por otro. Las implementaciones de intérpretes capaces de ofrecer esta abstracción se han denominado intérpretes metacirculares *(metacircular interpreter*s) [Wand88].

3-LISP

La idea de la torre infinita de intérpretes propuesta por Smith [Smith82] en el ámbito teórico tuvo distintas implementaciones [Rivières84]. El desarrollo de los prototipos de intérpretes metacirculares comenzó por la implementación de lenguajes de computación sencilla. El diseño de un intérprete capaz de ejecutar un número infinito de niveles computacionales, supone una elevada complejidad que crece a medida que aumentan las capacidades computacionales del lenguaje a interpretar.

El primer prototipo metacircular desarrollado, denominado *3-Lisp* [Wand88], interpreta un subconjunto del lenguaje *Lisp* [Steele90], y permite cosificar y reflejar un número indefinido de niveles computacionales. El estado computacional que será reflejado en este lenguaje está formado por tres elementos [Wand88]:

- Entorno *(environment)*: Identifica el enlace entre identificadores y sus valores en tiempo de ejecución.
- Continuación *(continuation)*: Define el contexto de control. Recibe el valor devuelto de una función y lo sitúa en la expresión que se está evaluando, en la posición en la que aparece la llamada a la función ya ejecutada.
- Almacén *(store)*: Describe el estado global de la computación en el que se incluyen contextos de ejecución e información sobre los sistemas de entrada y salida. De esta forma, el estado de computación de un intérprete, denominado metacontinuación *(metacontinuation)* [Wand88], queda definido formalmente por tres valores *(e, r, k)*, que podrán ser accedidos desde el metasistema como un conjunto de tres datos (cosificación).

La capacidad de representar formalmente y mediante datos el estado computacional de una aplicación en tiempo de ejecución aumenta en complejidad al aumentar el nivel de abstracción del lenguaje de programación en el que haya sido codificada. Debido a esto la mayoría de los prototipos de intérpretes metacirculares desarrollados computan lenguajes de semántica reducida.

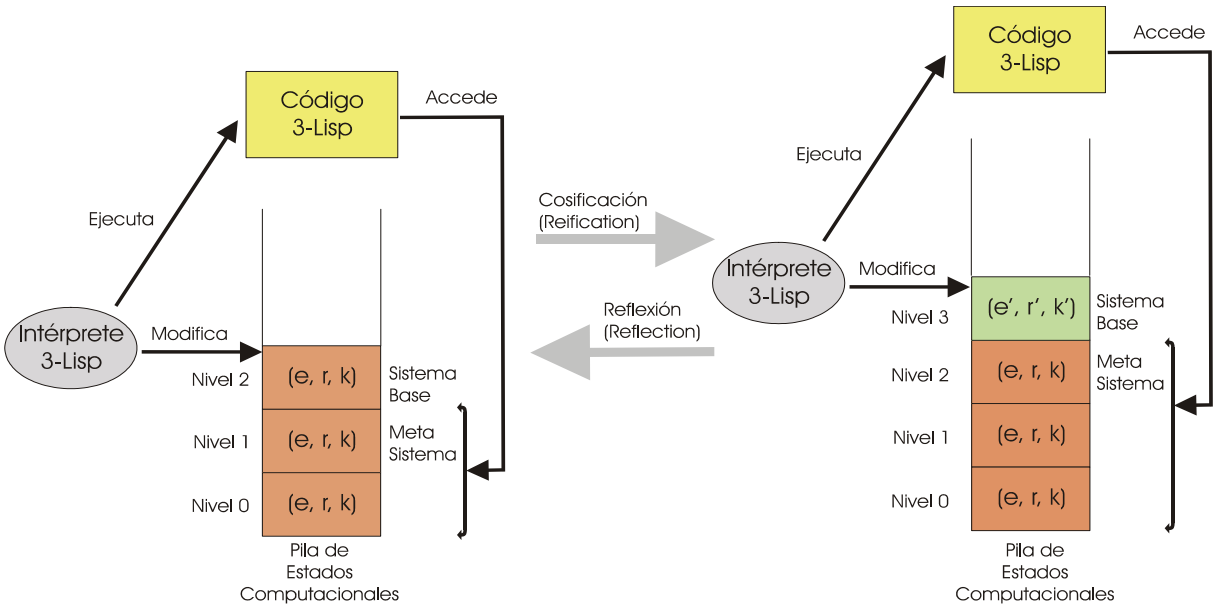

Figura 3.27: Implementación de un intérprete metacircular de *3-Lisp*

En la Figura 3.27 se aprecia el funcionamiento de los prototipos de interpretación de *3-Lisp*. Existe un programa codificado en *3-Lisp* que posibilita el cambio de nivel con las operaciones *reify* (aumento de nivel*)* y *reflect* (reducción de nivel). Un intérprete del subconjunto de *Lisp* definido va leyendo y ejecutando el lenguaje fuente. La ejecución de la aplicación supone la modificación de los tres valores que definen el estado computacional de la aplicación *(e, r, k)*. En la interpretación se puede producir un cambio de nivel de computación:

- Cosificación (*reify*): Se apila el valor del estado computacional existente *(e, r, k)* y se crea un nuevo estado de computación *(e', r', k')*. Ahora el intérprete trabaja sobre este nuevo contexto y la aplicación puede modificar los tres valores de cualquier nivel inferior como si de datos se tratase.
- Reflexión (*reflect*): Se desapila el contexto actual volviendo al estado anterior existente en la pila. La ejecución continúa donde había cesado antes de hacer la última cosificación.

Las condiciones necesarias para implementar un prototipo de estas características son básicamente dos:

- Expresividad computacional mediante un único lenguaje. Puesto que realmente existe un único intérprete, éste estará obligado a animar un único lenguaje de programación. En todos los niveles computacionales deberá utilizarse por tanto el mismo lenguaje de programación.
- Identificación formal del estado computacional. La semántica computacional del lenguaje deberá representarse como un conjunto de datos manipulables por el programa. La complejidad de este proceso es muy elevada para la mayoría de lenguajes de alto nivel.

Este lenguaje ha originado una serie de desarrollos, como *3-KRS* [Maes87b], que cambia el intérprete metacircular por una implementación basada en metaobjetos.

ABCL/R2

La familia de lenguajes *ABCL* fue creada para llevar a cabo investigación relativa al paralelismo y orientación a objetos. Inicialmente se desarrolló un modelo de computación concurrente denominado *ABCM/1 (An object-Based Concurrent computation Mode*l) y su lenguaje asociado *ABCL/1 (An object-Based Concurrent Language*) [Yonezawa90]. En la implementación de un modelo computacional concurrente, la reflexión computacional ofrece la posibilidad de representar la estructura y la computación concurrente mediante datos (cosificación) utilizando abstracciones apropiadas. En la definición del lenguaje *ABCL/R* (*ABCL* reflectivo) [Watanabe88], a partir de cualquier objeto "o" puede obtenerse su metaobjeto "O", que representa, mediante su estructura, el estado computacional de "o". Se implementa además un mecanismo de conexión causal para que los cambios del metaobjeto se reflejen en el objeto original. Esta operación puede aplicarse tanto a objetos como a metaobjetos, por lo que así se logra la implementación de una torre infinita de intérpretes (intérprete metacircular).

Para facilitar la coordinación entre metaobjetos del mismo tipo, y para definir comportamientos similares de un grupo de objetos, la definición del lenguaje *ABCL/R2* [Matsuoka91] añade el concepto de "metagrupo", consistente en un metaobjeto que define el comportamiento de un conjunto de objetos del sistema base. Para implementar

la adición de metagrupos, surgen determinadas ampliaciones del sistema:

- Nuevos objetos de núcleo *(kernel objects)* para gestionar los grupos: *The Group Manager*, *The Primary Metaobject Generator* y *The Primary Evaluator*.
- Se crea un paralelismo entre dos torres de intérpretes: la torre de metaobjetos y la torre de metagrupos.
- Objetos no cosificables *(non-refying objects)*. Se ofrece la posibilidad de definir objetos no reflectivos para eliminar la creación de su metaobjeto adicional y obtener así mejoras de rendimiento.

METAJ

MetaJ [Doudence99] es un prototipo que trata de ofrecer las características propias de un intérprete metacircular para un subconjunto del lenguaje de programación *Java* [Gosling96]. Creado en *Java*, el intérprete permite cosificar objetos para acceder a su propia representación interna (reflexión estructural) y a la representación interna de su semántica (reflexión computacional). Inicialmente el intérprete procesa léxica y sintácticamente el código fuente, creando un árbol sintáctico con nodos representativos de las distintas construcciones sintácticas del lenguaje. El método `eval` de cada uno de estos nodos representará la semántica asociada a cada uno de sus elementos sintácticos.

Conforme la interpretación del árbol se va llevando a cabo, se van creando objetos representativos de los creados por el usuario en la ejecución de la aplicación. Todos los objetos creados poseen el método `reify` que nos devuelve un metaobjeto, que es la representación interna del objeto que nos permite acceder a su estructura (atributos, métodos y clase), así como a su comportamiento (por ejemplo, la búsqueda de atributos o la recepción de mensajes). De esta forma se puede obtener:

- Reflexión estructural: Accediendo y modificando la estructura del metaobjeto, se obtiene una modificación estructural del objeto. Existe un mecanismo de conexión causal que refleja los cambios realizados en todo metaobjeto.
- Reflexión computacional: Se consigue modificando la clase de una instancia por una clase derivada que derogue el método que especifica la semántica a alterar.

La parte novedosa de *MetaJ* sobre el resto de sistemas estudiados a lo largo de este capítulo reside en la capacidad de poder cosificar metaobjetos en el grado que deseemos. Si invocamos al método `reify` de un metaobjeto, obtendremos la representación de un metaobjeto pudiendo modificar así la semántica de su comportamiento. El acceso reflectivo no posee un límite de niveles, constituyéndose así como un caso particular de un intérprete metacircular.

APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Los sistemas estudiados ofrecen el mayor nivel de flexibilidad computacional respecto al dominio de niveles computacionales a modificar. El acceso a cualquier elemento de la torre de intérpretes permitiría modificar la semántica del sistema en cualquier grado. No obstante, aunque teóricamente facilita la comprensión del concepto de reflexión, en un campo más pragmático puede suponer determinados inconvenientes. La posibilidad de acceso simultáneo a distintos niveles puede producir la pérdida del

conocimiento de la semántica del sistema, sin conocerse realmente cuál es el significado del lenguaje de programación [Foote90].

En los sistemas estudiados, se ha dado precedencia a ofrecer un número indefinido de niveles de computación accesibles frente a ofrecer un mayor grado de información a cosificar. Si tomamos *3-Lisp* como ejemplo, ofrece la cosificación del estado computacional de la aplicación, pero no permite modificar la semántica del lenguaje, ya que el intérprete es monolítico e invariable. Para conseguir este requisito mediante la implementación de infinitos niveles, debería especificarse la semántica del lenguaje en el propio estado de computación, extrayéndola del intérprete monolítico. Por otra parte, en el caso de *MetaJ* se restringen a priori las operaciones semánticas a modificar, puesto que han de estar predefinidas como métodos de una clase de comportamiento. Además, la modificación a llevar a cabo en el comportamiento ha de especificarse en tiempo de compilación.

Como conclusión, cabe mencionar que a la hora de desarrollar un sistema reflectivo, puede resultar más útil ahondar en la cantidad de información a cosificar y el modo en el que ésta pueda ser expresada que aumentar el número de niveles computacionales cosificables. Un punto adicional a destacar, propio del sistema *ABCL/R2*, es su definición de metagrupos. Este concepto se usa para agrupar el comportamiento de un conjunto de objetos en una sola abstracción. Para conseguirlo, se introducen un conjunto de entidades adicionales y dos torres de interpretación paralelas. Si bien la agrupación de metaobjetos puede ser atrayente para reducir la complejidad del metasistema, existen mecanismos más sencillos para conseguir dicha funcionalidad, como por ejemplo los objetos rasgo o *trait* de los sistemas orientados a objetos basados en prototipos, que ofrecen esta abstracción de un modo más sencillo.

3.4.5.3 MÁQUINA VIRTUAL NITRO

Esta plataforma, creada por Francisco Ortín Soler [Ortin01], utiliza distintas técnicas de reflexión para desarrollar un sistema computacional de programación extensible y adaptable dinámicamente, que no posea dependencia alguna de un lenguaje de programación específico, empleando una plataforma virtual heterogénea. Para ello se usa una máquina abstracta diseñada desde cero en lugar de usar una existente, por considerarse que no existe ninguna que siendo modificada pueda adaptarse a los requisitos planteados, procurando además que esta máquina posea un tamaño y complejidad semántica reducida.

La máquina implementará una serie de primitivas básicas de reflexión, que puedan servir como base computacional para el resto del sistema. Una máquina de carácter introspectivo y de un tamaño reducido como ésta permitiría además ser implantada fácilmente en entornos computacionales distintos, haciéndola efectivamente independiente de la plataforma. Este sistema permite extender su nivel de abstracción computacional de forma dinámica, utilizando para ello su propio lenguaje de programación, sin que haya que modificar la implementación reducida de la máquina virtual antes mencionada para ello, por lo que se mantiene su portabilidad (al no modificarse el código de la máquina, se mantienen todas sus versiones intactas, no hay que propagar cambios a todas ellas). Mediante este mecanismo el sistema permite diseñar abstracciones que soporten múltiples funcionalidades (como persistencia o distribución) que luego pasen a formar parte del sistema como ampliación a la funcionalidad básica ofrecida por el mismo. Estas abstracciones podrán además ser adaptadas a cualquier aplicación en tiempo de ejecución.

Mediante el empleo de las características reflectivas con las que cuenta la plataforma, se ha construido un sistema para procesar lenguajes de forma genérica, por lo que cualquier aplicación podrá interactuar con otra usando el modelo computacional de objetos ofrecido por la máquina abstracta, independientemente del lenguaje con la que

haya sido creada.

El objetivo de este sistema es pues lograr una flexibilidad dinámica superior a cualquiera de los actualmente existentes, tanto en expresividad como en los elementos computacionales que son adaptables en el mismo. Tanto la estructura de las aplicaciones en tiempo de ejecución como la especificación léxica, sintáctica y semántica de cualquier lenguaje de programación que se use serán parámetros configurables dinámicamente, tanto por la propia aplicación como por cualquier otro programa. El nivel de reflexión propuesto no tiene pues ninguna restricción respecto a las características computacionales a configurar, ni respecto al modo de expresar su adaptación, siendo además toda la flexibilidad dinámica: Las aplicaciones no tienen que finalizarse para adaptarlas a nuevos requisitos que surjan en tiempo de desarrollo u otros motivos que aconsejen cambios en el diseño de la aplicación ejecutada. Este sistema permite pues una reflexión dinámica de cualquier elemento computacional que se necesite, logrando además otro tipo de beneficios como la portabilidad. Pasaremos ahora a describirlo brevemente, para ver cómo se han logrado todas las características mencionadas anteriormente y su soporte para reflexión.

Capas del Sistema

El sistema está dividido en tres capas o elementos bien diferenciados, que pueden apreciarse en la figura 3.28.

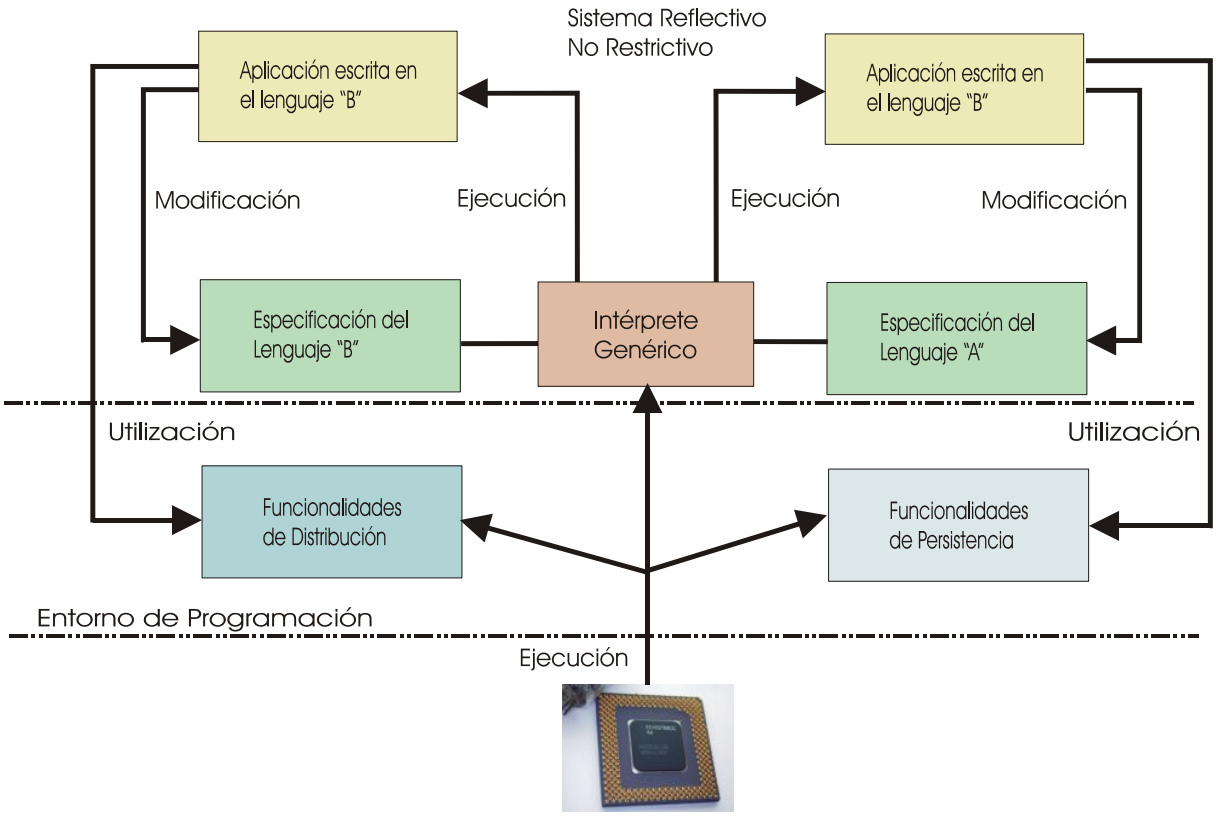

Figura 3.28: Arquitectura del sistema *nitrO*

- Máquina Abstracta: El motor computacional del sistema es la implementación de una máquina abstracta. Todo el código ejecutado por ésta es portable y, por tanto, independiente de la plataforma física empleada. El empleo de una máquina

abstracta para construir el sistema permite una serie de funcionalidades y beneficios que justifican su elección, tal y como se mencionará en un capítulo posterior. Además, de esta forma todo el sistema comparte el mismo modelo computacional, no dependiendo éste de ninguna plataforma física, por lo que es más fácilmente portable.

- Entorno de Programación: Sobre la máquina abstracta se desarrolla un código que facilite la labor del programador. Este código es portable, independiente del lenguaje (cualquier aplicación en el sistema, codificada en cualquier lenguaje, puede utilizarlo) e independiente de la plataforma. La máquina abstracta ha de poseer la característica de ser extensible para, sin necesidad de modificar su implementación, pueda desarrollarse sobre ella un entorno de programación con funcionalidades de un mayor nivel de abstracción, como distribución o persistencia, y que de esta forma no existan diferentes versiones de la máquina que soporten una serie de características diversas. Estas nuevas funcionalidades se desarrollarán empleando este código.
- Sistema Reflectivo No Restrictivo: Hasta esta tercera y última capa, todas las aplicaciones del sistema se desarrollan sobre el lenguaje nativo de la máquina abstracta. Este lenguaje posee una semántica fija para sus aplicaciones. Mediante esta capa se otorga independencia del lenguaje al sistema y flexibilidad dinámica, sin restricciones previas, de los lenguajes a utilizar.

Con esta estructura de capas definida, el funcionamiento queda definido de la siguiente forma: Un intérprete genérico toma la especificación de un lenguaje y ejecuta una aplicación codificada en éste. La aplicación puede hacer uso del entorno de programación e interactuar con otras aplicaciones codificadas en otros lenguajes. Adicionalmente, podrá modificar la especificación del lenguaje utilizado, reflejándose estos cambios en la adaptación de la semántica de la propia aplicación, de forma instantánea. Todo este proceso aparece reflejado en la figura 3.28.

Único Modelo Computacional de Objetos

Como ya se dijo anteriormente, el empleo de una máquina abstracta permite a este sistema tener un único modelo computacional de objetos. Todo código existente (sea o no de las aplicaciones de usuario que funcionan sobre la arquitectura de capas de la máquina antes mencionada) que esté vinculado al sistema se ejecutará sobre el modelo de objetos común soportado por la máquina virtual, lo que permitiría que las aplicaciones puedan interactuar entre sí independientemente de su lenguaje de programación, al "hablar" todas en los mismos "términos". La codificación del entorno de programación sobre el lenguaje propio de la máquina facilitará la utilización de éste sin dependencia alguna del lenguaje a utilizar.

Para que esto sea posible, una de las tareas a llevar a cabo por el intérprete genérico del sistema reflectivo consistiría en traducir las aplicaciones codificadas mediante un lenguaje de programación cualquiera, escogido por el programador, a su equivalente en el modelo computacional de la máquina abstracta. Una vez hecho esto, la aplicación podrá interaccionar con el resto del sistema como si hubiese sido codificada sobre su lenguaje nativo, ya que habrá sido traducida a una especificación completamente compatible con el resto de entidades existentes en dicho sistema.

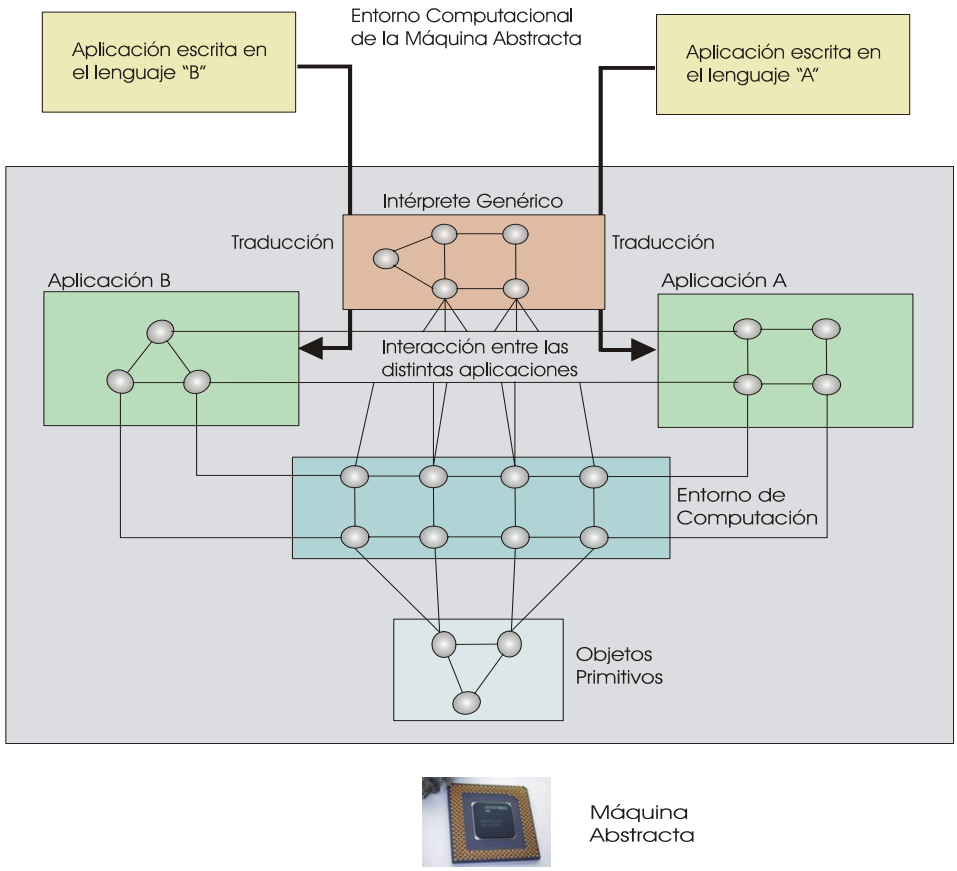

Figura 3.29: Interacción entre distintos objetos de un mismo espacio computacional

En la figura 3.29 puede verse cómo se estructura el sistema a la hora de que las aplicaciones interactúen. La máquina abstracta parte de un conjunto mínimo y cuidadosamente escogido de objetos primitivos, que serían su funcionalidad básica. Al ser de un tamaño reducido, portar la máquina a otras plataformas no resulta excesivamente complejo. Como con este conjunto mínimo de objetos la funcionalidad que la máquina ofrece a los programadores es escasa, a través de la extensibilidad de esta plataforma virtual se desarrolla código que eleva el nivel de abstracción en la programación de la plataforma, facilitando así la tarea del programador.

El resultado de hacer todo este proceso es un único espacio computacional de interacción de objetos que ofrecen sus servicios al resto del sistema. Como la máquina los computa de modo uniforme, el origen de dichos objetos es indiferente.

Máquina Abstracta

Como ya se ha dicho, la máquina abstracta supone el motor computacional del conjunto del sistema. La migración de éste a una plataforma consistiría simplemente en la recompilación de su implementación para el nuevo sistema nativo. El modelo computacional de objetos definido por ésta representará el propio del sistema y la forma en la que las distintas aplicaciones interactúen entre sí. Los objetivos generales conseguidos con la implementación de esta máquina abstracta son:

- Conjunto Reducido de Primitivas: Al necesitarse que la máquina abstracta pueda implantarse fácilmente en entornos de naturaleza heterogénea (en teoría cualquier

sistema computacional, aunque sea muy reducido, debería poder instalar una implementación del mismo), se reduce el número de primitivas computacionales de la misma, para facilitar así su migración.

- Mecanismo de Extensibilidad: Como ya se ha explicado, el hecho de que la máquina tenga un número de primitivas computacionales lo más reducido posible hace que, para que el programador pueda disponer de un mayor número de funcionalidades, sea necesario que la máquina abstracta posea un mecanismo de extensibilidad que permita añadirle más primitivas. Ha de tenerse en cuenta que realizar estas ampliaciones mediante la inclusión de instrucciones nuevas para ampliar la funcionalidad podría generar diversas implementaciones diferentes de la misma máquina, hacer que el código ya no fuera portable o aumentar la complejidad de tal manera que se perjudique su implantación en entornos heterogéneos. Por ello se ha determinado que el método más óptimo para esta tarea es codificar las ampliaciones en el propio lenguaje de programación de la máquina, a partir de sus primitivas computacionales básicas. Al ser código propio de la máquina, cualquier funcionalidad adicional implementada será portable a cualquier otra plataforma logrando los dos objetivos deseados:

    - Que la máquina virtual siga siendo sencilla y por tanto fácilmente portable a cualquier plataforma.
    - Que se logre un mayor nivel de abstracción dotando al programador de un mayor número de herramientas con las que hacer sus programas.

En esta figura se ve cómo se ha llevado a cabo en el sistema ese mecanismo de extensibilidad, mediante la creación de una segunda capa llamada entorno de programación, siguiendo los principios dados.

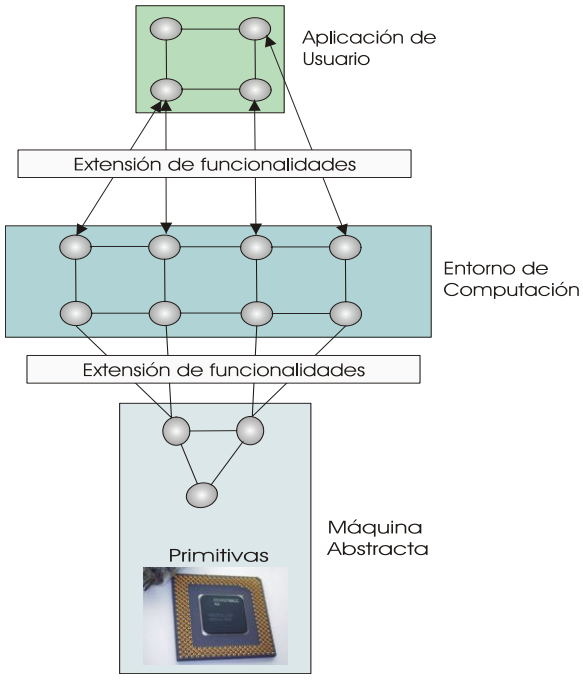

Figura 3.30: Extensibilidad de las funcionalidades primitivas de la m. abstracta

Entorno de Programación

El *software* flexible desarrollado sobre la plataforma abstracta y enfocado a elevar el nivel de abstracción del sistema tiene un conjunto de objetivos básicos:

- Portabilidad e Independencia del Lenguaje: La codificación del entorno de programación es independiente de toda plataforma física y lenguaje de programación, de manera que sólo tendrá que codificarse una única vez. Del mismo modo, cualquier aplicación desarrollada sobre cualquier lenguaje puede utilizar los servicios ofertados por este código. La única tarea a tener en cuenta a la hora de implantar el entorno de programación en una plataforma es la selección del subconjunto de funcionalidades que deseemos instalar. En función de las necesidades del sistema, de su potencia de cómputo y de la memoria disponible, el sistema demandará una parte o la totalidad de las funcionalidades del entorno de programación.
- Adaptabilidad: El entorno de programación aumenta el nivel de abstracción del sistema, ofreciendo nuevas funcionalidades adaptables. Si el entorno de programación ofrece funcionalidades de persistencia, éstas son flexibles respecto al entorno físico utilizado para almacenar los objetos. Además, si el programador desea introducir su propio sistema de persistencia, su diseño debería estar enfocado a minimizar el número de pasos necesarios para implantarlo. La adaptabilidad también se aplica a las primitivas de la máquina abstracta, siendo el objetivo buscado que la totalidad de las funcionalidades ofrecidas por esta capa del sistema sean adaptables a las distintas necesidades de los programadores.
- Introspección: La implantación del entorno de programación en sistemas computacionales heterogéneos requiere el conocimiento dinámico del subconjunto de funcionalidades instaladas en cada plataforma. Si una plataforma no ofrece las características de persistencia por limitaciones de espacio, el código de usuario deberá tener la posibilidad de consultar si esta funcionalidad ha sido implantada antes de intentar hacer uso de ella.

Sistema Computacional Reflectivo sin Restricciones

Apoyándose en la definición de un sistema reflectivo como aquél que puede acceder a niveles computacionales inferiores dentro de su torre de interpretación [Wand88], este sistema justifica la necesidad de una altura de dos niveles –interpretación de un intérprete– y una anchura absoluta –acceso al metasistema no restrictivo [Ortin01]. Esto quiere decir que sólo son necesarios dos niveles de interpretación, pero que el nivel superior ha de poder modificar cualquier característica computacional del nivel subyacente que le da vida.

Comparando las características de este sistema frente aquellas que ofrece uno basado en un *MOP*, podemos decir que:

- No es necesario estipular lo que va a adaptarse previamente a la ejecución de la aplicación.
- Existe un mecanismo de expresividad en el que cualquier característica del sistema pueda adaptarse dinámicamente.
- El sistema es totalmente independiente del lenguaje.
- Los grados de flexibilidad que consigue son: introspección, estructural, semántica

y lenguaje.

El esquema que sigue este sistema se muestra en la siguiente figura:

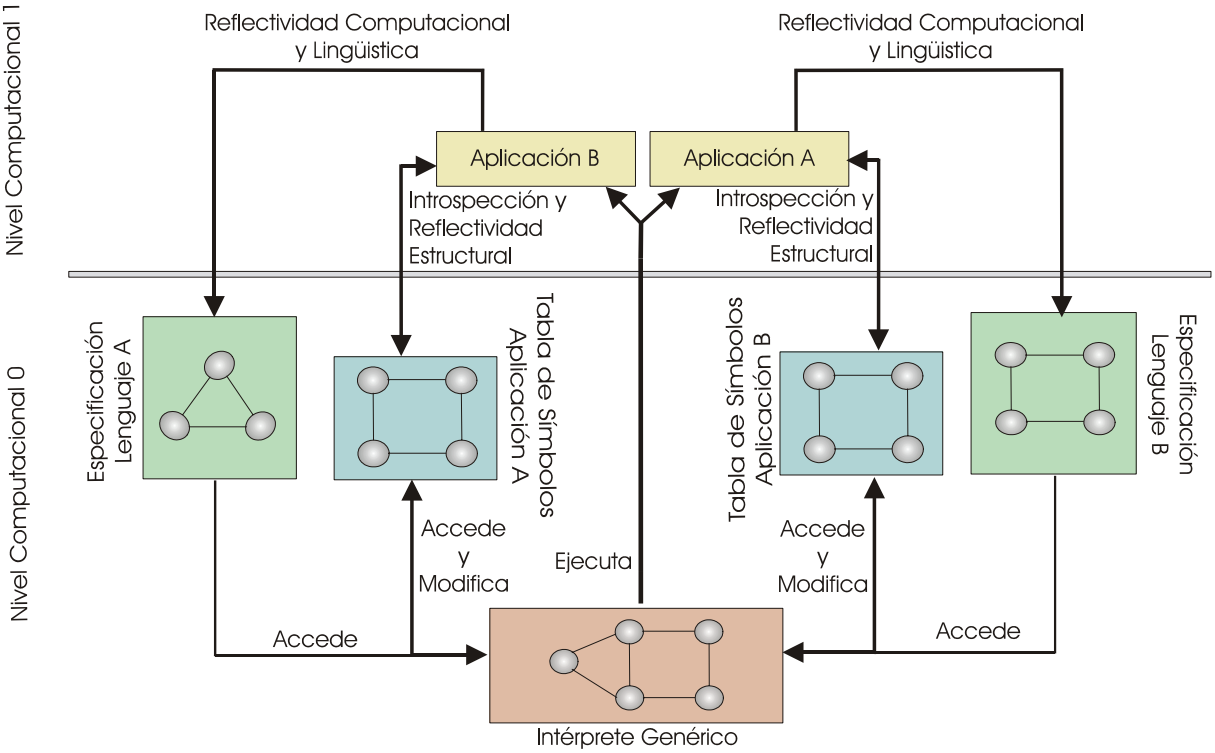

Figura 3.31: Esquema general del sistema computacional no restrictivo

Una de las diferencias frente a los intérpretes convencionales es la creación de un intérprete genérico independiente del lenguaje. Este procesador ejecuta cualquier aplicación escrita sobre cualquier lenguaje de programación. De este modo, la entrada a este programa no se limita, como en la mayoría de los intérpretes, a la aplicación a ejecutar, sino que está parametrizado además con la especificación del lenguaje en el que dicha aplicación haya sido codificada.

Implementando un intérprete genérico en el que las dos entradas son la aplicación a procesar y la especificación del lenguaje en el que ésta haya sido escrita, se consigue hacer que el sistema computacional sea independiente del lenguaje. Para cada aplicación evaluada, el intérprete genérico deberá crear un contexto de ejecución o tabla de símbolos dinámica, en el que aparezcan todos los objetos creados a raíz de ejecutar el programa. Se trata de ubicar todos los objetos creados en el contexto de ejecución de la aplicación en un único espacio de nombres.

La flexibilidad del sistema se obtiene cuando, en tiempo de ejecución, la aplicación accede a la especificación de su lenguaje, o bien a su tabla de símbolos existente. El resultado de estos accesos supone distintos grados de flexibilidad:

- Si analiza, sin modificar, su tabla de símbolos supondrá introspección.
- En el caso de modificar los elementos de su tabla de símbolos, la flexibilidad obtenida es reflectividad estructural.
- El acceso y modificación de la semántica propia de la especificación de su lenguaje de programación significará reflectividad computacional.

- La modificación de las características léxicas o sintácticas de su lenguaje producirán una adaptación lingüística.

El modo en el que las aplicaciones de usuario accedan a su tabla de símbolos y a la especificación de su lenguaje no posee restricción alguna. La horizontalidad de este acceso ha de ser plena. Para conseguir lo propuesto, la principal dificultad radica en la separación de los dos niveles computacionales mostrados en la figura anterior. El nivel computacional de aplicación poseerá su propio lenguaje y semántica, mientras que el nivel subyacente que le da vida –el intérprete– poseerá una semántica y lenguaje no necesariamente similar. El núcleo de este sistema se centra en un salto computacional del nivel de aplicación al de interpretación.

Para que una aplicación pueda modificar, sin restricción alguna, el intérprete que lo está ejecutando, la raíz computacional de este sistema está soportada por una implementación, *hardware* o *software*, de una máquina abstracta, como se vio anteriormente. Sobre su dominio computacional, se desarrolla el intérprete genérico independiente del lenguaje de programación. La especificación de cada lenguaje a interpretar se llevará a cabo mediante una estructura de objetos, representativa de su descripción léxica, sintáctica y semántica. Este grupo de objetos sigue el modelo computacional descrito por la máquina y, por tanto, pertenece también a su espacio computacional.

En la ejecución de una aplicación por un intérprete, éste siempre debe mantener dinámicamente una representación de sus símbolos en memoria. El intérprete genérico ofrecerá éstos al resto de aplicaciones existentes en el sistema. De esta forma, por cada aplicación ejecutada por el intérprete, se ofrecerá una lista de objetos representativos de su tabla de símbolos. Siguiendo este esquema, y como se muestra en la figura siguiente, el dominio computacional de la máquina abstracta incluye el intérprete genérico y, por cada aplicación, el conjunto de objetos que representan la especificación de su lenguaje y la representación de sus símbolos existentes en tiempo de ejecución.

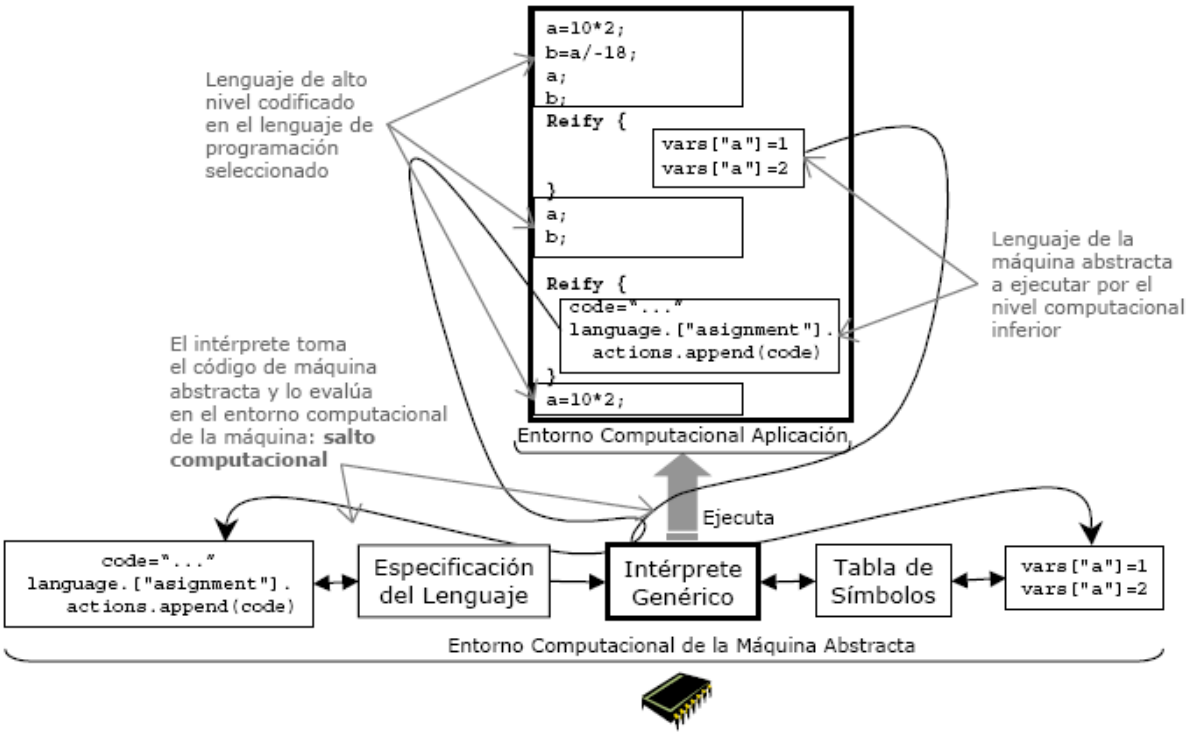

Figura 3.32: Salto computacional producido en la torre de intérpretes

En el espacio computacional del intérprete se encuentran las aplicaciones de usuario codificadas en distintos lenguajes de programación. Cualquiera que sea el lenguaje de programación utilizado, una aplicación de usuario tiene siempre una instrucción de acceso al metasistema (`reify`). Dentro de esta instrucción –entre llaves– el programador podrá utilizar código de su nivel computacional inferior: código propio de la máquina abstracta. Así, una aplicación poseerá la expresividad de su lenguaje más la propia de la máquina abstracta.

Siempre el que intérprete genérico analice una instrucción `reify` en un programa de usuario, en lugar de evaluarla como una sentencia propia del lenguaje interpretado, seguirá los siguientes pasos:

- Tomará la consecución de instrucciones codificadas en el lenguaje de la máquina abstracta, como una cadena de caracteres.
- Evaluará o descosificará los datos obtenidos, para que sean computados como instrucciones por la máquina abstracta.

El resultado es que el código de usuario ubicado dentro de la instrucción de cosificación es ejecutado por el nivel computacional inferior al resto de código de la aplicación. Es en este momento en el que se produce un salto computacional real en el sistema.

Para que la implementación del mecanismo descrito anteriormente sea factible, la posibilidad de evaluar o descosificar dinámicamente cadenas de caracteres como computación es uno de los requisitos impuestos a la máquina abstracta. Además, la interacción directa entre aplicaciones es otro de los requisitos impuestos a la plataforma virtual en la descripción global de la arquitectura del sistema. Cualquier aplicación, desarrollada en cualquier lenguaje, podrá interaccionar directamente –sin necesidad de una capa *software* adicional– con el resto de aplicaciones existentes. De este modo, el código de la aplicación propio de la instrucción `reify`, al ser ejecutado en el entorno computacional de la máquina, podrá acceder a cualquier aplicación dentro de este dominio, y en concreto a su tabla de símbolos y a la especificación de su lenguaje:

- Si analiza, mediante la introspección ofrecida por la máquina abstracta, su propia tabla de símbolos, estará obteniendo información acerca de su propia ejecución: introspección de su propio dominio computacional.
- Si modifica la estructura de alguno de sus símbolos, haciendo uso de la reflectividad estructural de la máquina, el resultado es reflectividad estructural de su nivel computacional.
- La modificación, mediante la utilización de la reflectividad estructural de la máquina, de las reglas semánticas del lenguaje de programación supone reflectividad computacional o de comportamiento.
- Si la parte a modificar de su lenguaje es su especificación léxica o sintáctica, el resultado obtenido es reflectividad lingüística del nivel computacional de usuario.

Vemos como otros dos requisitos necesarios en la máquina abstracta para llevar a cabo los procesos descritos son introspección y reflectividad estructural de su dominio computacional.

Finalmente comentaremos que la evaluación de una aplicación debe realizarse por el intérprete genérico analizando dinámicamente la especificación de su lenguaje, de forma que la modificación de ésta conlleve automáticamente al reflejo de los cambios

realizados. De este modo, no es necesaria la implementación de un mecanismo de conexión causal ni la duplicación de información mediante metaobjetos, puesto que el intérprete ejecuta la aplicación derogando parte de su evaluación en la representación de su lenguaje.

El salto computacional real ofrecido por este sistema cobra importancia en el momento en el que la aplicación de usuario accede a las estructuras de objetos representantes de su lenguaje de programación o de su tabla de símbolos. Estos lenguajes están representados de manera que sea posible su manipulación, gracias a la reflectividad estructural de la máquina abstracta. No obstante, no profundizaremos más en la descripción de las características de este sistema, ya que consideramos que se ha mostrado una panorámica lo suficientemente amplia de sus características, arquitectura y posibilidades. La descripción completa de todos los aspectos de *nitrO* puede obtenerse de [Ortin01].

Aportaciones y Carencias:

Mediante esta breve exposición de las características de este sistema, hemos visto cómo su diseño está especialmente cuidado para proporcionar al programador un grado de flexibilidad completo y sin restricciones, contando con un nivel de adaptabilidad más avanzado que el existente en otros sistemas que pretenden conseguir objetivos similares. Este sistema posee los niveles de reflexión más elevados, soportados mediante el especial diseño de su máquina abstracta y de todo el sistema en general, sin que existan a priori restricciones acerca de lo que puede ser reflejado. Además, otra serie de características como su portabilidad y la interoperabilidad completa de las aplicaciones que sobre el mismo se ejecutan, hacen que su implementación, aunque esté en fase de prototipo, sea una vía de estudio interesante para investigar avances en todas estas materias y las ideas descritas en su diseño pueden servir como base para la realización de trabajos relacionados.

Por otra parte, debe mencionarse también que el prototipo existente del sistema posee un bajo rendimiento, debido a la carencia de la máquina de técnicas de optimización dinámica de código como *JIT*, técnicas que serán descritas en un capítulo posterior.

3.5 CONCLUSIONES

A lo largo de este capítulo se ha visto cómo la reflexión es una técnica que puede ser empleada para obtener un alto grado de flexibilidad en un sistema computacional. Para elaborar las conclusiones, se dividirán dichos sistemas reflectivos teniendo en cuenta una serie de criterios enumerados a continuación.

3.5.1 Momento en el que se Produce el Reflejo

La reflexión en tiempo de ejecución otorga un elevado grado de flexibilidad al sistema, puesto que éste pueda adaptarse a contextos no previstos en fase de diseño. Cuando una aplicación necesita poder especificar nuevos requisitos dinámicamente, la

reflexión en tiempo de compilación no es suficiente. Sin embargo la reflexión estática o en tiempo de compilación posee una ventaja sobre la dinámica: la eficiencia de las aplicaciones en tiempo de ejecución. La adaptabilidad dinámica de un sistema produce una pérdida de rendimiento del mismo en su ejecución.

También hemos podido constatar cómo los sistemas estáticos ofrecen reflexión del lenguaje de programación, cuando esto no ocurre en ningún sistema dinámico de los vistos, manteniéndose éste inamovible.

3.5.2 Información Reflejada

El primer nivel de información a reflejar es la estructura del sistema en un modo de sólo lectura: introspección. La utilidad práctica de este nivel de reflexión queda patente por el número de sistemas comerciales que la utilizan. Mediante su uso se permite desarrollar fácilmente sistemas de componentes, de persistencia, comprobaciones de tipo dinámicas, o *middlewares* de distribución, entre otras.

El segundo grado de flexibilidad es la reflexión estructural, en la que se permite tanto el acceso como la modificación dinámica de la estructura del sistema. A nivel práctico existen muchas posibilidades para este tipo de sistemas, muchas de ellas todavía no explotadas completamente. Ejemplos pueden ser interfaces gráficas adaptables mediante la incrustación, eliminación y modificación dinámica de la estructura de los objetos gráficos, aplicaciones de bases de datos que trabajen con un abanico de información adaptable en tiempo de ejecución, o la apertura a un nuevo modo de programación adaptable dinámicamente [Golm98] y creación de nuevos patrones de diseño [Ferreira98].

Cuando la semántica del sistema puede modificarse nos encontramos en el tercer nivel de esta clasificación: reflexión computacional. Ésta última ofrece una flexibilidad elevada para todo el sistema en su conjunto. Se ha utilizado en la mayoría de casos a nivel de prototipo, aplicándose a depuradores *(debugger*s), compilación dinámica (*JIT*, *Just In Time compilatio*n), desarrollo de aplicaciones en tiempo real, o creación de sistemas de persistencia y distribución.

No obstante, a pesar de que este tercer nivel proporciona un nivel de flexibilidad superior, no está exento de inconvenientes. Por una parte la mayoría de los sistemas vistos ofrecen limitaciones respecto al grado de modificación de la semántica adaptable, y a la imposibilidad de modificar el propio lenguaje de programación, aspectos ambos que ha sido solucionados en algunos trabajos como [Ortin01], pero que reflejan también la complejidad que supondría incorporar esta capacidad a sistemas ya existentes sin que dicha incorporación suponga una ruptura del sistema con el código ya desarrollado para el mismo.

3.5.3 Niveles Computacionales Reflectivos

En la torre de intérpretes enunciada por Smith [Smith82] analizada anteriormente, el acceso de un sistema a su metasistema se representaba como el salto al intérprete que ejecutaba dicha aplicación. En un sistema reflectivo podemos preguntarnos cuántos niveles computacionales necesitamos y qué ganaríamos introduciendo más. Coincidiendo con la mayoría de los autores, un sistema altamente flexible y manejable es aquél que ofrece una elevada "anchura" y no "altura" de su torre de intérpretes. Esto quiere decir que es más útil obtener una forma sencilla y potente de

modificación del metasistema desde el sistema base ("anchura" de la torre) que la capacidad de modificar el comportamiento del lenguaje que especifica el comportamiento, denominado metacomportamiento, que sería la "altura" de la torre. Como se ha dicho anteriormente, contar con la posibilidad de acceder un número infinito de niveles de computación puede hacer al programador perder la semántica real del sistema con el que está trabajando.

4 LENGUAJES ORIENTADOS A OBJETOS BASADOS EN PROTOTIPOS

En este capítulo se describirá un paradigma alternativo a la orientación a objetos basada en clases para la construcción de *software* orientado a objetos: La orientación a objetos basada en prototipos. Este capítulo está basado en [Ortin01] y [Cardelli96]. En el paradigma alternativo mencionado, la abstracción de objeto, que es la entidad básica en el mismo, queda definida en función de una serie de primitivas de reflexión estructural:

- **Un objeto se define como un conjunto de miembros *(slots*)**. Éstos pueden constituir datos representativos del estado dinámico del objeto (atributos) o su comportamiento (métodos). La diferencia entre los dos tipos de miembros es que los segundos pueden ser evaluados, describiendo la ejecución de un comportamiento.
- **Adición dinámica de miembros**. Todo objeto posee un miembro computacional primitivo capaz de añadirle dinámicamente un `slot` de cualquier tipo.
- **Eliminación dinámica de miembros**. Igual que el anterior, mediante la primitiva `removeSlot`.

Por tanto, en función de la noción utilizada para agrupar objetos de igual comportamiento en un modelo computacional orientado a objetos, es posible establecer la siguiente clasificación de lenguajes [Evins94]:

- Lenguajes orientados a objetos basados en clases.
- Lenguajes orientados a objetos basados en prototipos.

El modelo computacional de objetos basado en clases se apoya en la agrupación de objetos de igual estructura y comportamiento. Todos los objetos que cumplan con unas características comunes serán instancias de la misma clase [Booch94]. Se establece así una relación de instanciación entre objetos y clases, de manera que no será posible crear y utilizar un objeto si no se ha definido previamente la clase a la que pertenece. La estructura estática de un objeto y su comportamiento en función de su estado, están enteramente definidos por su clase, mediante sus atributos y métodos respectivamente. Cuando se crea un objeto como instancia de una clase, su estado estará definido por el conjunto de valores correspondientes a los atributos de la estructura definida por su clase, pudiendo variar éste conjunto dinámicamente.

En el modelo computacional basado en prototipos no existe el concepto de clase, sino que la única abstracción existente es el objeto [Borning86]. Es el objeto el que describe su estructura (conjunto de atributos), su estado (los valores de éstos) y su comportamiento (la implementación de los métodos que pueda interpretar), es decir, el objeto es el elemento central de este modelo. Estudiaremos las diferencias y semejanzas existentes entre ambos, y también sus ventajas e inconvenientes.

4.1 TIPOS DE HERENCIA EN MODELOS DE CLASES Y PROTOTIPOS

4.1.1 Herencia en Modelos Basados en Clases

Antes de ver aspectos más concretos acerca de cómo se organizan y crean los objetos en el modelo basado en prototipos, conviene hacer una descripción acerca de las relaciones de herencia existentes entre objetos en este modelo, comparadas con las existentes en el modelo basado en clases.

En un **modelo basado en clases**, los objetos se crean a partir de la estructura especificada por las mismas a través de *new*, y típicamente se permiten operaciones como acceso/modificación de atributos e invocación de métodos. Una posible primera aproximación a cómo se representaría esta estructura es la siguiente:

```
class Mamifero
{
        String nombre;

        public String getNombre() {   return nombre; }
        public void setNombre(String nombre) { this.nombre = nombre; }
}
```

Figura 4.1: Modelo inicial de cómo se pueden guardar miembros de un objeto

No obstante, como quiera que en este modelo los objetos instancia de una clase C comparten la implementación de los métodos de la misma y guardan una información de estado privada para cada uno de ellos, el siguiente modelo parece más adecuado:

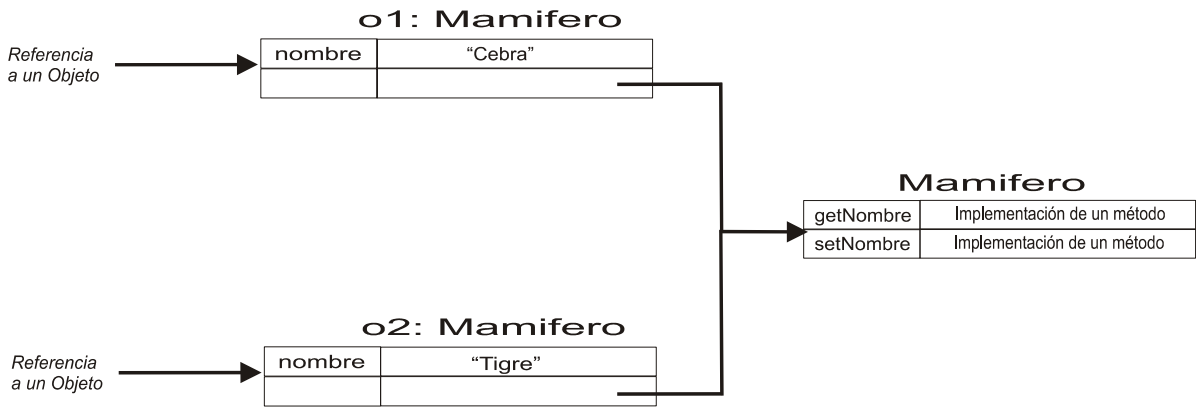

Figura 4.2: Modelo alternativo de cómo se pueden guardar miembros de un objeto

En el primer modelo los atributos y los métodos están integrados en el objeto, mientras que en el segundo la implementación de los métodos está delegada en otra entidad (la clase). En cualquier caso, para un lenguaje orientado a objetos basado en clases ambas formas de guardar los miembros son equivalentes. Cuando se va a hacer la invocación de un método (por ejemplo `o.getNombre()`) ocurre una búsqueda del mismo para ejecutar su código. Aunque esta búsqueda dependería de los detalles concretos de cómo se guardan los miembros en memoria, en este tipo de lenguajes esos detalles se ocultan al usuario, dando siempre la impresión de que es el objeto sobre el que se llama el método el propietario del mismo, y de que éstos por tanto están integrados en él.

En el momento de introducir la herencia en el modelo, aparecerán pues dos aproximaciones a la hora de cómo contener la información del nuevo objeto heredado. Entendiendo la herencia como una extensión de las características de un objeto padre por parte de un objeto hijo, los posibles modelos usados para guardar la información serían los siguientes:

```
class Mamifero
{
        String nombre;
        public String getNombre() { return nombre; }
        public void setNombre(String nombre) { this.nombre = nombre; }
}
class Perro extends Mamifero
{
        int nPatas = 4;
        public String ladrar () { return "wof, wof"; }
}
```

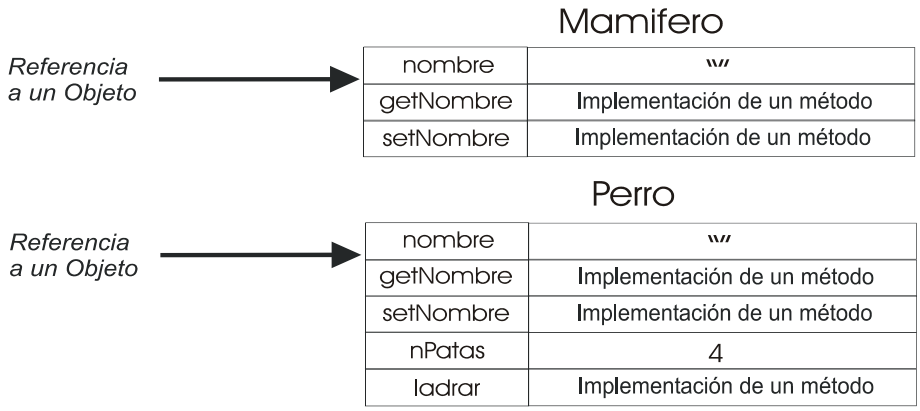

Figura 4.3: Modelo inicial de cómo se pueden guardar miembros heredados en una clase

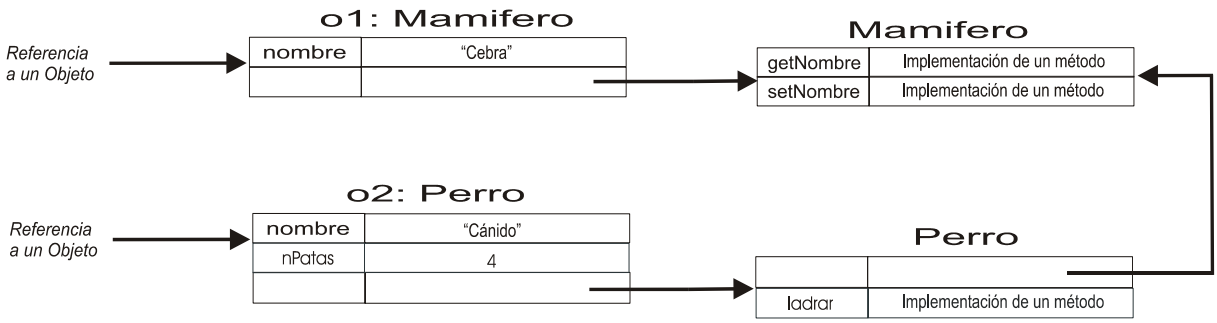

Figura 4.4: Modelo alternativo de cómo se pueden guardar miembros heredados en una clase

Vemos pues como nuevamente en una de las opciones cada objeto guarda todos sus miembros (propios y heredados) mientras que en el segundo caso se usa delegación. No obstante, en lenguajes basados en clases de tipos estáticos, los heredados de las clases padre se integran dentro de la estructura de la clase que los hereda [Cardelli06]:

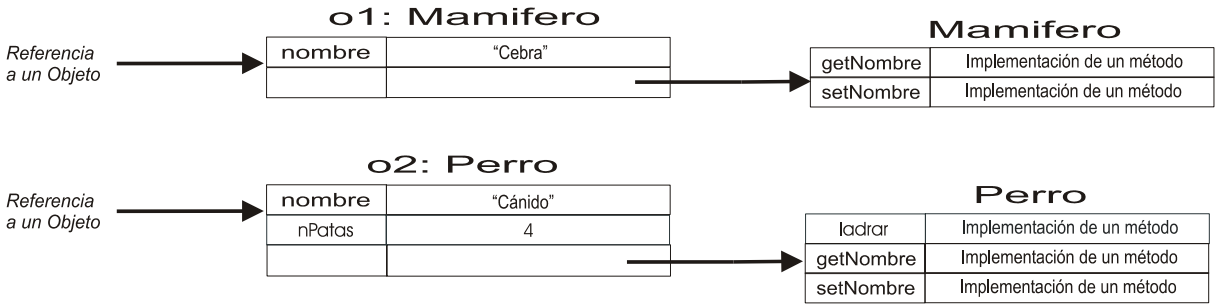

Figura 4.5: Guardar miembros heredados en un lenguaje basado en clases con tipos estáticos

De esta forma, cualquier subclase que redefina un método de alguna de sus clases padre, guardaría por tanto el método redefinido (con su nuevo código) dentro de la estructura asociada a ella misma. Este modelo de herencia se denomina herencia por concatenación, como veremos posteriormente. Como se ha dicho antes, a la hora de acceder a los miembros se crea la ilusión de que todos son propiedad del objeto sobre el que se llama, y no podemos acceder a los mecanismos y estructuras internas que se encargan de determinar como se guardan los atributos o métodos para manipularlos directamente. Los lenguajes basados en prototipos que veremos a continuación sí permitirían acceder y manipular de alguna forma estas estructuras y por tanto crean más posibilidades a la hora de implementar la herencia.

4.1.2 Herencia en Modelos Basados en Prototipos

En un **lenguaje basado en prototipos** esta forma de estructurar la información cambia de manera que puedan adquirir una serie de características y propiedades que se describirán a lo largo del capítulo actual. Como veremos posteriormente, en este segundo modelo los objetos se crean mediante la clonación de un objeto prototipo, que

al igual que una clase también describe objetos, pero éste a su vez es también un objeto. La forma de plantear la herencia en un lenguaje de este tipo es similar a la que hemos visto en los lenguajes basados en clases: se trata de "reutilizar" miembros de otros objetos[8], lo que requiere tener una estrategia de obtención de los miembros de los objetos "padre" y otra de incorporación de esos miembros a los objetos hijos. Llegados a este punto, se plantean pues cuatro posibles alternativas de funcionamiento de la herencia de objetos:

- Los miembros pueden ser obtenidos de las objetos padre (o "donantes"), de forma **implícita** o **explícita** [Cardelli96].
- Los miembros heredados pueden ser **integrados** (o **concatenados**, usando la terminología del lenguaje *Kevo* [Cardelli96]) en el objeto "receptor" (con lo cual pasan a formar parte de él) o bien **delegados** al objeto padre o "donante" (con lo que se accederán a través de indirecciones desde el objeto "receptor", manteniéndose sólo en el objeto "donante"). En el primer caso hablamos de **herencia por concatenación**, en el segundo hablamos de **herencia por delegación**.

En cualquiera de los casos, el receptor de los métodos siempre es el objeto a través del cual se llama a los mismos (`this`, `self` o cualquier palabra reservada que se use para referirse a él por el lenguaje concreto). Mientras que en la herencia por concatenación los "receptores" son independientes de los "donantes", en el segundo pueden llegar a formarse complejas redes de dependencia entre objetos.

En la herencia implícita, uno o varios "donantes" son designados y todos los atributos susceptibles de ello son heredados implícitamente. En cambio en la herencia explicita se seleccionan un conjunto de atributos de forma explícita (de ahí el nombre) para que sean "donados" al "receptor". En este último caso los conceptos de `super` y `override` no tienen sentido. Ambos tipos de herencia son extremos y es posible encontrar soluciones intermedias, como la llamada *mixin inheritance*, que designa una colección de atributos a "donar" con un nombre concreto, pero que no llegan a formar un objeto completo.

En la **herencia por concatenación** (*concatenation* o *embedding inheritance* [Cardelli96]), los miembros de los objetos "receptores" heredados son copias de los "donantes" y la invocación de un método heredado funciona de forma idéntica a la invocación del método en el objeto "donante".

8 *Nótese que ya no usamos el término "clase". Como se menciono anteriormente, los lenguajes basados en prototipos no tienen ese concepto.*

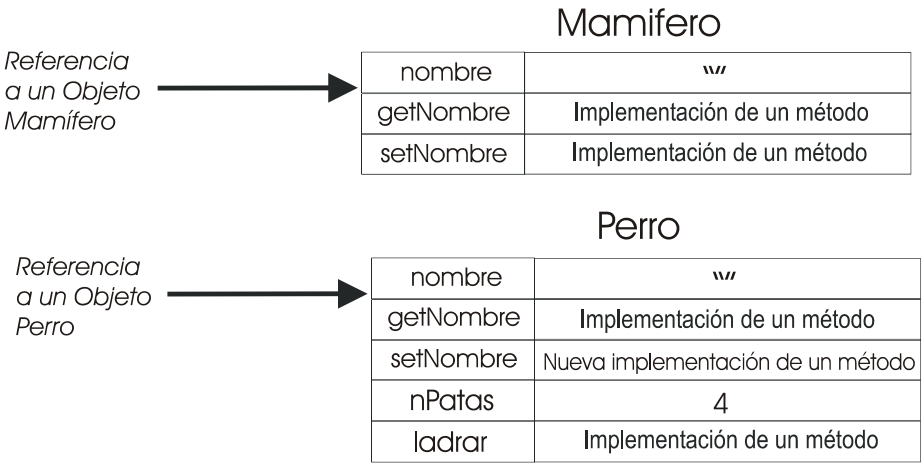

Figura 4.6: Herencia por concatenación

Este modelo de herencia ha sido adoptado por lenguajes basados en prototipos como *Kevo* [Taivalsaari93] y *Obliq* [Cardelli06] y puede ser tanto implícita (extender copias de objetos existentes con nuevos atributos) o explícita (reensamblar partes de objetos existentes para formar nuevos objetos). Una de las principales diferencias de este modelo con el que vamos a describir a continuación es que si por ejemplo se modificase un método, esta modificación sólo afectaría al objeto sobre el que se realizó, y no podría afectar a otros objetos de la jerarquía (lo que no ocurre en delegación).

En la **herencia por delegación** pues los objetos contienen enlaces a los miembros de los objetos "donantes", redigiriendo los accesos a atributos y las invocaciones a métodos a otros objetos, de manera que el primer objeto (el "receptor" de miembros) es una extensión del segundo (el "donante"). A los lenguajes basados en prototipos que permiten esta operación se dice que están basados en delegación. También esta herencia puede ser explícita o implícita. En cualquiera de los casos, si se invoca un método `m()` cualquiera, puede ocurrir que éste se encuentre en un objeto distinto del que se está usando para llamar al método, pero siempre se aplica el código de `m()` sobre el objeto actual (dentro del cuerpo de `m()`, `self` o `this` siempre será el objeto que se ha usado para llamar a `m()`).

La herencia por delegación explicita nuevamente designa un conjunto de miembros que serán "donados" a otros objetos, que establecerán un enlace con los mismos en su propietario original. La herencia por delegación implícita en cambio especifica que lo que se "dona" son objetos completos, de manera que todos los objetos hijos a uno dado compartirían su estado. Es posible no obstante, crear copias en cada objeto individual de aquellos atributos que no deseemos compartir con los demás, de manera que un acceso a un atributo de este tipo siempre se hará sobre la copia local. Esta figura muestra un esquema sencillo de la herencia por delegación:

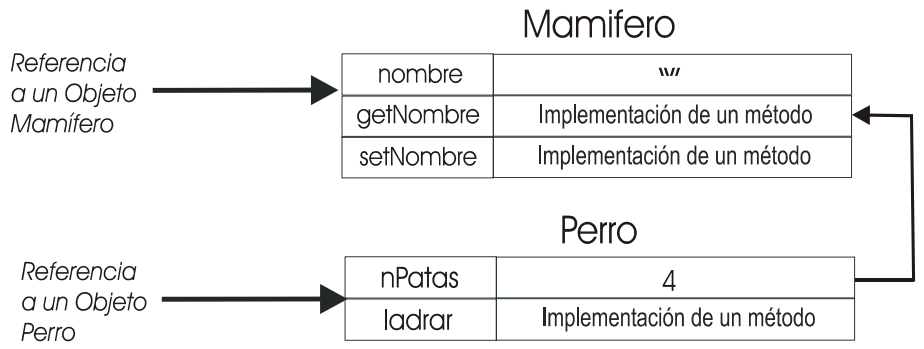

Figura 4.7: Ejemplo de herencia por delegación

Por tanto, en un lenguaje que siga esta aproximación podemos esperar que cuando se le envíe un mensaje a un objeto, se analizará si éste posee un método que lo implemente, ejecutándolo si existe. En caso contrario, el proceso se repite en todos los objetos de su jerarquía de herencia, si el objeto original tuviese objetos padre, hasta encontrar el miembro buscado, o devolver un error si no es posible encontrarlo.

Este tipo de herencia se complementa con el concepto de **herencia dinámica**. La herencia es llamada estática cuando los atributos heredados son siempre los mismos (es decir, no se pueden cambiar en tiempo de ejecución). Lo contrario se denomina herencia dinámica, y permite tanto añadir, borrar o modificar cualquier miembro que se hereda. Con este tipo de herencia es posible también cambiar un conjunto de atributos heredados por otro, o el objeto padre de uno dado en cualquier momento. Para permitir esto, el mecanismo del paso de mensajes basado en delegación envía ascendentemente los mensajes si el objeto no puede responderlos, recorriendo la jerarquía de objetos existente en cada momento. Por ello, para que la herencia dinámica sea coherente, el modelo de herencia ha de estar basado en delegación.

Esto hace que la herencia dinámica en lenguajes orientados a objetos basados en prototipos sea una asociación más, dotada de una semántica adicional. En el caso del lenguaje de programación *Self* [Ungar87], la identificación de esta semántica especial es denotada por la definición del miembro `parent`, que es realmente el objeto padre a uno dado. Por tanto, al tratarse la herencia como una asociación, sería posible modificar en tiempo de ejecución este objeto, obteniendo así el mencionado mecanismo de herencia dinámica [Taivalsaari92], que es mucho más flexible y carece de los problemas planteados por el mecanismo de herencia estático típico de lenguajes basados en clases [Holub03].

A modo de resumen podemos decir entonces que en la herencia por concatenación un objeto nuevo contiene copias de los miembros de todos aquellos objetos designados como "donantes" del mismo. El acceso a los atributos del "donante" es idéntico al acceso a los mismos sobre el objeto "receptor", y normalmente es un acceso más eficiente que en el otro tipo de herencia. No obstante, esta estrategia requiere un espacio en memoria bastante más grande (salvo que se usen optimizaciones), dada la duplicidad de miembros que puede ocurrir. En cambio, en la herencia por delegación el objeto contiene enlaces a los objetos "donantes" externos y durante la invocación de métodos los accesos a los atributos siempre se hacen sobre los que posea el objeto que se use para invocar dicho método, aunque esto puede hacer que la implementación del lenguaje sea más compleja.

Mientras que en los lenguajes orientados a clases ambas aproximaciones son casi siempre equivalentes [Cardelli96], en lenguajes basados en prototipos sí puede establecerse una clara distinción entre ambos. En el modelo de delegación los donantes pueden contener miembros que se pueden modificar y los cambios se verán desde los objetos que heredan de los mismos, lo que facilitaría las operaciones de reflexión. No obstante, tanto la concatenación como la delegación son dos formas diferentes de conseguir herencia con prototipos.

A modo de conclusión de este punto, podemos decir que las principales ventajas de la herencia por delegación son que permite aprovechar mejor el espacio en memoria, al compartir miembros entre objetos, y que se muestra superior a la hora de permitir cambios en la estructura de objetos, al tener una estructura más flexible. Por otra parte, es cierto que puede crear un conjunto de dependencias entre objetos que pueden ser muy complejas y aumentar de este modo la fragilidad del *software*. Esto no ocurre con concatenación, ya que los objetos son autónomos.

4.2 ORGANIZACIÓN DE LOS ELEMENTOS EN EL MODELO BASADO EN PROTOTIPOS

La eliminación del concepto de clase en el modelo de prototipos hace que la implementación de características reflectivas sea más sencilla, al no existir un elemento externo al propio objeto que condicione su estructura y comportamiento, siendo el propio objeto el que guarda toda esa información. De todas formas, en este modelo sigue siendo posible, si se desea, agrupar comportamientos similares de diferentes objetos, de forma similar a lo que hacen las clases en la *POO* clásica. Para ello, se usaría un objeto concreto que agruparía comportamientos, poseyendo pues solamente métodos (también se admiten atributos estáticos, compartidos por todas las instancias asociadas). Estos objetos describirán entonces el comportamiento común de todos los objetos que estén asociados al mismo (nunca tienen información de estado) y se denominan objetos de característica o rasgo (*trait objects*) [Lieberman86]. En la Figura 4.8, el objeto *trait* llamado `Object` define el comportamiento `toString` de todos los objetos. Del mismo modo, `Point` define el comportamiento de sus dos objetos asociados.

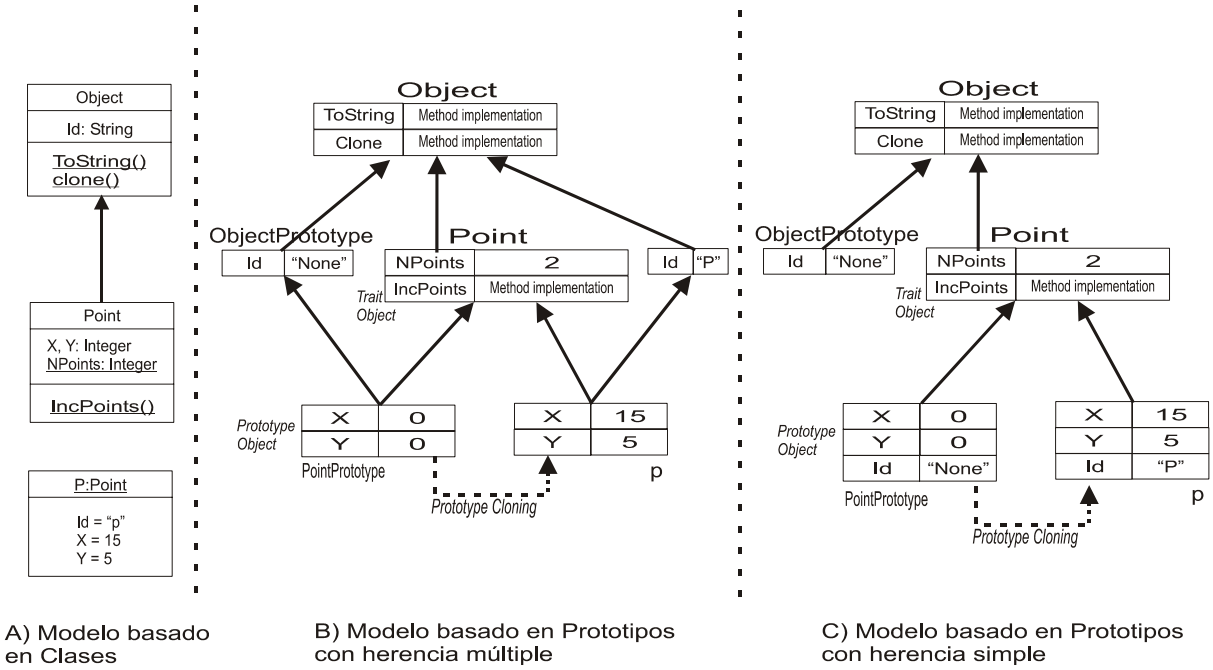

Figura 4.8: Representación de clases y objetos en los dos modelos.

Del mismo modo que se han agrupado objetos de igual comportamiento se podrá también agruparlos por estructura. Un prototipo (*prototype*) será el objeto descriptor de la estructura común, utilizado para hacer copias exactas de él (clonaciones). En este modelo computacional, mediante la utilización de prototipos y la primitiva de clonación, se obtiene una funcionalidad equivalente a la instanciación o creación de objetos a través de una clase en su modelo homólogo. En la Figura 4.8, la creación de un punto pasa por la clonación de su prototipo. Éste posee la estructura común de todos los puntos (atributos `x` e `y`), su estado inicial (ambos valores iguales a cero) y el comportamiento común definido por el objeto *trait* `Object`.

Por tanto, en este tipo de lenguajes sin clases tanto los *traits* como objetos *prototype* son objetos normales, pero con roles especiales. En concreto, podemos

esperar el siguiente comportamiento "típico" por parte de los elementos de este modelo (aunque puede haber algunas variantes en función de la implementación concreta del lenguaje):

- Los *traits* están únicamente pensados para ser "padres" compartidos por los objetos ordinarios y no deberían ser usados ni clonados directamente.
- Los *prototypes* están pensados sólo para ser generadores de objetos vía clonación. Tampoco deberían ser usados directamente o clonados.
- Los objetos "normales" están pensados para contener estado de forma local, deberían usar *traits* para sus métodos.

Estas restricciones pueden ser o no forzadas por la implementación. No obstante, esta separación de roles en principio viola el "espíritu" original de los lenguajes orientados a prototipos, ya que existirán objetos que no pueden funcionar por si solos. Sin embargo, esta separación permite emular el comportamiento de los lenguajes basados en clases, es decir, la separación entre *traits* y prototipos puede formar una estructura muy similar a lo que hacen los lenguajes basados en clases tradicionales. Podemos decir por tanto que las clases son sólo una de las posibles formas de crear objetos que tengan miembros en común, y que éstos mecanismos presentados ofrecen más opciones para ello y por tanto más flexibilidad.

El hecho de que estos lenguajes puedan emular los conceptos de los lenguajes basados en clases hace que muchos lenguajes dinámicos, aunque su modelo computacional este basado en prototipos, usen la palabra reservada `class` (por ejemplo *Smalltalk*, *Python* o *Ruby*, todos ellos descritos en esta tesis). Estos lenguajes no las implementan como los lenguajes basados en clases (las clases no representan comportamientos y estructura compartidos), sino que simplemente modelan *trait objects* (comportamiento compartidos). Los objetos son responsables de guardar su propia estructura y pueden tener comportamientos específicos (métodos propios, privados a la instancia).

Como ejemplos de lenguajes orientados a objetos basados en prototipos, además de los mencionados, podemos mencionar algunos como *Self* [Ungar87] [Chambers89], *Moostrap* [Mulet93], *ObjectLisp*, *Cecil* [Chambers93], *NewtonScript* [Swaine94], *PROXY* [Leavenworth93], o *Visual Zero* [Schofield06].

4.3 UTILIZACIÓN DE LENGUAJES ORIENTADOS A OBJETOS BASADOS EN PROTOTIPOS

A partir de la información dada en los apartados anteriores, en este apartado resumiremos las peculiaridades y características existentes en el modelo orientado a objetos basado en prototipos referentes a su utilización, destacando cuales son las principales novedades que aporta y sus efectos.

4.3.1 Reducción Semántica

Una de las características destacables en el modelo de prototipos frente al de clases reside en la simplicidad del primero, obtenida al:

- Eliminar el concepto de clase. No es necesario crear una clase para todos los objetos que necesitemos.
- Suprimir la dependencia necesaria existente entre un objeto y una clase. En el modelo de clases, siempre ha de existir entre ellos una relación de instanciación, y el estado del objeto ha de ser coherente con la estructura definida por su clase. Ahora ya no será necesario.
- Eliminar la comprobación estática de tipos. En ambos modelos un objeto sólo puede recibir los mensajes implementados por su clase y superclases. En un sistema basado en clases esta comprobación es realizada en tiempo de compilación. En cambio, en el caso del modelo basado en prototipos, al constituirse la herencia como un mecanismo dinámico (delegación), no es posible llevar a cabo esta validación estáticamente, por lo que esta comprobación se hace de forma dinámica[9].

4.3.2 Inexistencia de Pérdida de Expresividad

Como hemos visto, la sencillez de este modelo computacional, basado únicamente en objetos, no conlleva que haya una pérdida en la expresividad de los lenguajes que lo utilizan. De hecho, se han realizado estudios que demuestran que la utilización de prototipos no supone pérdida alguna de expresividad [Ungar91] [Evins94], y que toda semántica representable mediante con el modelo de clases puede ser traducida al modelo de prototipos. Su demostración ha quedado patente en la implementación de compiladores de lenguajes basados en clases, como *Self* y *Java*, sobre la plataforma *Self* que utiliza prototipos [Wolczko96].

Para hacer una pequeña muestra de cómo se hace esta traducción de un modelo a otro, mostraremos un esquema que ilustre cómo se expresarían una serie de objetos y clases en un modelo y en otro, como se ve en la figura 4.9.

9 Es por esta razón por la que los sistemas basados en prototipos utilizan un sistema dinámico de tipos [Chambers89], o bien carecen de su inferencia [Cardelli97].

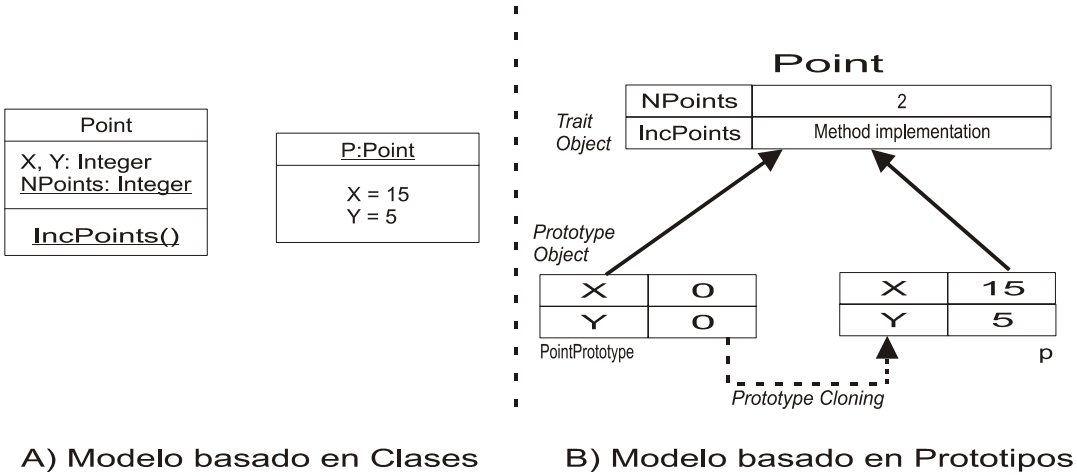

Figura 4.9: Miembros de clase en ambos modelos orientados a objetos.

Como se muestra en la figura anterior, la clase `Point` posee un atributo de clase (subrayado) que cuenta el número de instancias creadas, y un método de clase (también subrayado) que permite incrementarlo. La ubicación de ambos en el modelo de prototipos se sitúa en el objeto *trait*, al considerarse miembros compartidos por todos los objetos asociados. Éste será el que reciba los mensajes de que posea la clase y su es el que determinará el resultado de su ejecución. Por tanto, la utilización de este servicio se demandará mediante el *trait* (objeto `Point`) y no mediante una de sus instancias. Por último, vemos cómo los objetos punto son creados mediante la clonación del prototipo correspondiente, que posee la estructura de todos estos objetos, y también cómo cada objeto está asociado con el *trait object* correspondiente para poder acceder a su comportamiento.

4.3.3 Traducción Intuitiva de Modelos

Como se acaba de mostrar, la traducción del modelo basado en clases al modelo que utiliza prototipos se produce de un modo bastante intuitivo para el programador. Por esta traducción entre modelos intuitiva, la sencillez del modelo de prototipos y la carencia de pérdida en su expresividad, Wolczko incluso propone los prototipos como modelo universal de computación de lenguajes orientados a objetos [Wolczko96]. Llevando a cabo la traducción de cualquier lenguaje a un único modelo, la interacción intuitiva entre aplicaciones se podría lograr independientemente del lenguaje que haya sido utilizado para su construcción.

4.3.4 Coherencia en Entornos Reflectivos

Como muestra de las ventajas en cuanto a flexibilidad y posibilidades de modificación que tiene el modelo basado en prototipos, describiremos a continuación las limitaciones al respecto que tiene el modelo de clases y su solución. Dos problemas existentes en el campo de las bases de datos, y que también se plantean a la hora de hacer modificaciones a la estructura de clases y objetos en el modelo de objetos basado en clases, son la evolución y mantenimiento de versiones de esquemas. Pueden definirse de la siguiente forma [Roddick95]:

- Evolución de esquemas (*schema evolution*): Capacidad de una base de datos para permitir modificaciones en su diseño sin incurrir en pérdida de información.
- Mantenimiento de versiones de esquemas (*schema versioning*): Se produce cuando, tras haberse llevado a cabo un proceso de evolución del esquema, la base de datos permite acceder a la información tanto de un modo retrospectivo como mediante la estructura actual, manteniendo así toda la información para cada versión de esquema existente.

Con la aparición de las bases de datos orientadas a objetos este problema se traslada al concepto de clase empleado en un programa. Como ya se ha mencionado, el esquema (estructura y comportamiento) de un objeto queda determinado por la clase de la que es instancia (sería la entidad que contendría su diseño). En este caso se plantea el problema de qué modificaciones se deben efectuar en los objetos si se hacen cambios en su clase, es decir, si ocurre una evolución de esquemas. Esta cuestión aparecerá cuando tratemos de hacer modificaciones mediante el empleo de primitivas de reflexión estructural a cualquier clase que ya tenga instancias creadas, ya que habrá que determinar qué hacer con los objetos si se añade, elimina o modifica un miembro a su clase. Una posible solución consistiría en implementar un mecanismo que de forma ansiosa (modificar inmediatamente todas las instancias de un objeto dado en cuanto se altere la clase asociada a las mismas) o perezosa (modificar sólo aquellas instancias que traten de acceder a la información que ha sido modificada) se encargue de actualizar la estructura de las instancias para hacerlas coincidir con la de su clase y no romper así los principios del modelo computacional.

Mientras que el primer problema tiene una solución factible, el problema del versionado de esquemas no tiene una solución fácil en el modelo de objetos basado en clases, ya que no es nada sencillo buscar un modelo que permita que un objeto tenga modificaciones en su estructura particulares y privadas, al entrar entonces éste en conflicto con su clase por tener ambos elementos distinta estructura (se rompe la coherencia del modelo). Una de las posibles soluciones aportadas es la empleada por el sistema *MetaXa* antes visto (empleando clases sombra), que tiene la ventaja de no tener que prescindir del concepto de clase. No obstante, también quedó patente en su descripción que esta solución no resulta operativa al introducir demasiada complejidad al sistema. Incluso los propios autores del sistema, en [Golm97c], mencionan que sería más sencilla su implementación si se hiciese uso del modelo basado en prototipos.

Por tanto, modificar la estructura de instancias y objetos empleando el modelo de clases tradicional plantea muchos problemas, siendo esta modificación más sencilla de implementar en un modelo basado en prototipos, al no generar incoherencias en dicho modelo. Un ejemplo de ello es el sistema *Moostrap* [Mulet93]. Los distintos escenarios de modificación que pueden plantearse son:

- La modificación de la estructura de un único objeto se haría modificándolo directamente, ya no hay entidades externas que especifiquen su estructura y no existirían problemas de versionado de esquemas.
- La modificación de la estructura de los nuevos objetos que se puedan crear se podría llevar a cabo mediante la modificación del prototipo utilizado, a partir del cual se generan.
- La modificación del comportamiento compartido por una serie de objetos se conseguiría mediante la manipulación del objeto *trait* correspondiente, al que estarían asociados todos los objetos a manipular, y que verían pues los cambios en el comportamiento inmediatamente, produciéndose una evolución de esquemas.

En la figura 4.10 puede verse un ejemplo de cómo se harían las modificaciones mencionadas. En ella, podemos ver cómo la modificación (en gris) del *trait object* `Perro` hace que todos los objetos asociados al mismo adquieran el nuevo comportamiento, mientras vemos además cómo todos los objetos admiten modificaciones (todas en gris) que afectarán únicamente a esos objetos, ya sean estos atributos simples, complejos, otros objetos (`Caseta`), métodos, etc.

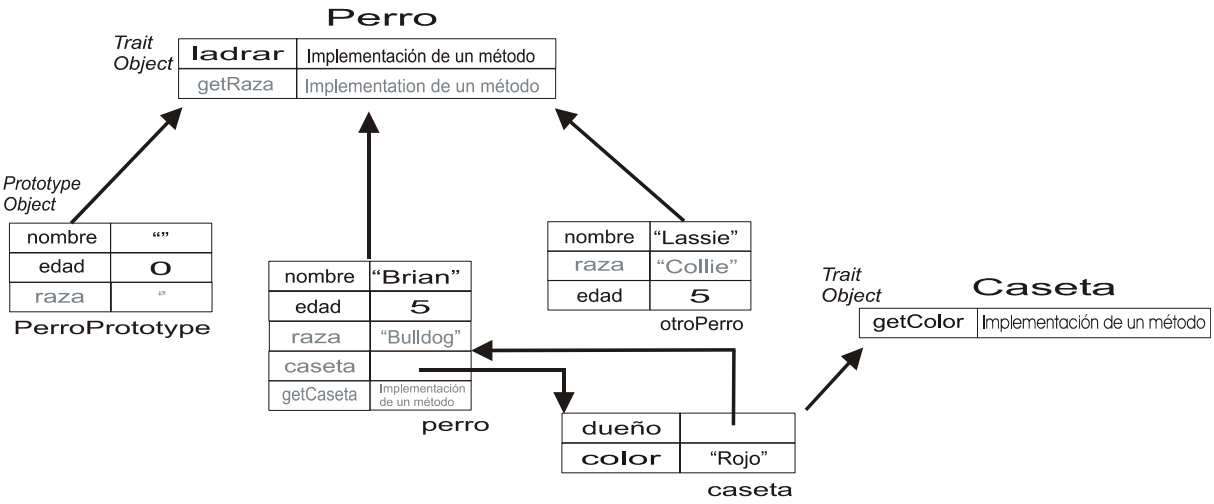

Figura 4.10. Modificaciones a objetos en el modelo orientado a prototipos

En este modelo, es el programador el que lleva el peso de agrupar los objetos por comportamiento y estructura. Esta agrupación puede cambiarse dinámicamente, ya que la herencia y las relaciones son también dinámicas, siendo una de sus principales ventajas que se puede identificar fácilmente lo que realmente se desea modificar (el grupo, uno en concreto o los sucesivos).

4.4 CONCLUSIONES

En función de la clasificación establecida al comienzo de este capítulo, veíamos cómo los modelos computacionales orientados a objetos podían utilizar clases o simplemente objetos. Como veíamos en el punto anterior, la utilización del modelo computacional basado en prototipos aporta un conjunto de ventajas. Sin embargo, los lenguajes que utilizan clases también poseen virtudes:

- El nivel de abstracción ofrecido por las clases es más elevado, permitiendo teóricamente al programador expresar un modelo más cercano al problema a resolver.
- La agrupación obligada de objetos mediante la definición de clases exige al programador la división de las aplicaciones a desarrollar en abstracciones a modelar, lo que puede ser conveniente en algunos casos, pero resta flexibilidad al programador.
- La definición de los tipos de objetos mediante sus clases permite detectar en tiempo de compilación errores de tipo, reduciendo así del número de errores

producidos en tiempo de ejecución y facilitando teóricamente la labor del desarrollador. No obstante, estos errores producidos por la posesión de un sistema de tipos dinámico pueden ser minimizados mediante las herramientas de prueba adecuadas.

En función del estudio realizado de ambos modelos, se puede afirmar como conclusión que los sistemas basados en clases están más orientados a la programación, mientras que la utilización de prototipos está más acorde con el desarrollo de una plataforma computacional. De cara a la modificación dinámica de la estructura de los elementos de una aplicación, en este capítulo ha quedado claro que sólo mediante el modelo de prototipos podrán hacerse las modificaciones necesarias de una forma coherente y sencilla, sin romper ninguna norma del modelo computacional usado ni tener que hacer soluciones de compromiso o aumentar la complejidad del sistema innecesariamente. Por otra parte, usar un modelo basado en prototipos no causa ninguna pérdida de expresividad puesto que, como se ha visto a lo largo de este capítulo, el modelo de prototipos puede implementar el modelo de clases, mientras que lo contrario no es posible.

5 LENGUAJES DINÁMICOS

Dado que el trabajo que se plantea en esta tesis está fundamentalmente orientado a dar un soporte más eficiente a lenguajes dinámicos, en este capítulo se hará un estudio de los lenguajes dinámicos más representativos que podemos encontrar en la actualidad. Aunque existen varias definiciones acerca de lo que se puede o no se puede considerar un lenguaje dinámico, en este trabajo se usará la utilizada en [Ortin05] por describir todos sus aspectos de forma completa. Según esta definición, un lenguaje dinámico es *"aquel lenguaje que permite la exploración y la adaptación de su estructura, comportamiento y entorno, soportando características que permitan que se modifique a si mismo y también la generación dinámica de código"*. La definición de lenguaje dinámico dada contempla el concepto de reflexión mencionado en un capítulo anterior, al mencionarse explícitamente la capacidad de modificar la estructura y/o comportamiento del lenguaje. Así, con un lenguaje dinámico podríamos realizar operaciones como las siguientes:

- Cambiar dinámicamente la interfaz de cualquier objeto y clase, alterando pues su conjunto de métodos y atributos y adaptándolos a nuevos requisitos.
- Modificar la implementación de cualquier método escogido en tiempo de ejecución.
- Adaptar sobre la marcha la estructura de una instancia concreta, sin que ello afecte necesariamente a otras instancias.
- Generar dinámicamente nuevo código a voluntad.
- Modificar el tipo de un objeto, cambiando pues completamente aquél que tuviera anteriormente durante la ejecución de un programa.
- Modificar el comportamiento o la estructura de un conjunto determinado de clases, usando el concepto de metaclase.

La naturaleza particular de estos lenguajes les confiere a su vez una serie de características comunes que veremos en el siguiente punto. Haremos una clasificación de los lenguajes dinámicos y describiremos algunos de los más utilizados, incidiendo especialmente en aspectos como su modelo computacional de objetos y sus capacidades flexibles, aspectos más afines a los objetivos a conseguir en esta tesis. Para ello, en la primera parte de este capítulo describiremos sus características generales y posteriormente mostraremos los lenguajes más representativos en las dos categorías en las que los hemos dividido: lenguajes de *scripting* y lenguajes de propósito general.

5.1 CARACTERÍSTICAS GENERALES DE LOS LENGUAJES DINÁMICOS

Como hemos dicho, los lenguajes dinámicos son aquéllos que permiten examinar y efectuar modificaciones sobre su propia estructura, comportamiento y entorno, con

características que le permiten su automodificación y la generación dinámica de código. Hoy en día es posible encontrar distribuciones muy completas de los mismos, válidas para desarrollar muchos tipos de aplicaciones diferentes, en paralelo con lenguajes estáticos, como *C#* y *Java*, que gozan de gran popularidad. Ejemplos son lenguajes como *Python*, *Ruby* o *Dylan*, que serán descritos en este capítulo.

Estos lenguajes ofrecen una serie de capacidades que proporcionan por tanto un alto grado de flexibilidad a los programas que se desarrollan con ellos, y les permiten responder mejor ante la aparición de nuevos requisitos o reglas de negocio, algo que sería más costoso de obtener mediante el empleo de lenguajes "estáticos" tradicionales como *C* y *C++*. Debido a los beneficios que ofrecen, los lenguajes dinámicos son usados en diferentes escenarios de la ingeniería del *software*, como:

- **Desarrollo de software con programación orientada a aspectos (*Aspect Oriented Programming* o *AOP*) y separación de incumbencias (*Separation of Concerns* o *SoC*)**: Como se ha mencionado en el capítulo anterior, el desarrollo de *software* con aspectos es una técnica que permite modularizar aspectos ortogonales de los sistemas *software*, siguiendo el principio de la separación de incumbencias [Hürsch95]. Este principio se basa en el hecho de que la orientación a objetos a veces no ofrece mecanismos lo suficientemente potentes para permitir una correcta modularización y reutilización del código de un programa, ya que algunas incumbencias (persistencia, *log*,...) tienen que estar entremezcladas y repartidas sobre varios módulos. Por tanto, los principales beneficios de un *software* construido mediante *AOP* son la posibilidad de alcanzar una mejor legibilidad, reusabilidad, mantenibilidad y productividad en el desarrollo del código [Laddad02].

  Como hemos visto, los sistemas *AOP* que ofrecen adición, modificación y eliminación de aspectos mientras el programa esté en ejecución se denominan sistemas de *AOP* dinámicos. La mayoría de estos sistemas se aprovechan de que lenguajes dinámicos ofrecen los servicios de la *AOP* dinámica como parte de su semántica [Ortin04], como *Phytius* [Pythius06] o *Lightweight Python AOP* (*Python*), *AOP/ST* (*Smalltalk*) [Bollert99] y *AspectOrientedRuby* (*Ruby*) [AORuby06].
- **Desarrollo de aplicaciones *Web***: Los lenguajes dinámicos permiten la manipulación de componentes *software* mientras la aplicación está en funcionamiento, usando sus capacidades de metaprogramación. Esta característica es muy importante en el desarrollo *Web*, donde es frecuente encontrarse con la necesidad de añadir nuevos requisitos en tiempo de ejecución. Con estos servicios, se sigue el principio *DRY* (*Don't Repeat Yourself*) [Venners03]. *DRY* establece que cada parte de la base de conocimiento de un sistema debe tener sólo una representación única y no ambigua. Esto reduce la repetición de código y el solapamiento de la información, reduciendo a su vez el coste de mantenimiento e incrementando la productividad. También puede disminuir el impacto de cambios realizados en ciertas partes de un programa, como el hecho de que un cambio en una capa de la aplicación (por ejemplo modificar el diseño de una tabla en la base de datos) pueda producir cambios importantes en otras capas de la misma [Venners03]. Dada la flexibilidad ofrecida por los lenguajes dinámicos, *Java* incorporara en su versión 1.6 un *framework* estándar para permitir ejecutar sobre esta plataforma lenguajes dinámicos de *scripting*, así como acceder a la propia plataforma [JSR06]. Otros ejemplos de uso de lenguajes dinámicos para el desarrollo *Web* es el *framework Ruby on Rails,* usado para crear aplicaciones *Web* cuya información resida en una base de datos [Thomas05], y *AJAX* (*Asynchronous JavaScript And XML*), una técnica de desarrollo *Web* que facilita la construcción de aplicaciones interactivas de esta clase [Crane05].
- **Modelos de objetos adaptativos**: Un modelo de objetos adaptativo es un

sistema que representa clases, atributos y relaciones como metadatos [Yoder01]. Los usuarios cambian los metadatos (el modelo de objetos) para reflejar los cambios en domino, y esos cambios modificarán el comportamiento del sistema. El modelo de objetos se guarda en una base de datos, siendo éste interpretado posteriormente. Consecuentemente el modelo de objetos es activo, de forma que, si se modifica, el sistema cambiará inmediatamente para reflejar estos cambios. Aunque los modelos de objetos adaptativos pueden ser desarrollados usando lenguajes estáticos como *Java*, para ello es necesario emplear diferentes patrones de diseño (*Type Object*, *Property* o el patrón *Strategy*), así como una gran cantidad de código adicional, para emular primitivas de reflexión estructural [Yoder01]. Dado que los lenguajes dinámicos ya ofrecen estos servicios como parte de su semántica, son un medio que facilita notablemente la creación de *software* que use estos modelos de objetos adaptativos.

- ***Frameworks* de aplicaciones**: La capacidad de inspeccionar y modificar la estructura y comportamiento de los programas, así como la creación de código nuevo en tiempo de ejecución para adaptar o decorar componentes que ya existen, hace que los lenguajes dinámicos sean una herramienta adecuada para desarrollar *frameworks* aplicados a dominios específicos. Usando estas capacidades, el usuario podrá crear código que adapte el comportamiento de un programa a unas necesidades determinadas. Ejemplos de esto son *Zope* [Zope06] (*software* empleado para la construcción de aplicaciones personalizadas y sistemas de gestión de contenidos), *Twisted* (un *framework* para desarrollar aplicaciones de red basado en eventos) [Fetting05] y *Peak*, un *software* de desarrollo de aplicaciones empresariales [Peak06].
- **Aplicaciones comerciales**: Los lenguajes dinámicos también tienen su aplicación en el desarrollo de determinados tipos de *software* comercial, en escenarios concretos como herramienta para la creación de rutinas de inteligencia artificial, *kits* de *modding* de algunos videojuegos [Firaxis05] o incluso como lenguaje de desarrollo de programas como *BitTorrent* [BitTorrent06], un popular *software P2P*. También podemos ver un ejemplo de uso en el *software* de modelado y renderizado 3D *Blender* [Blender06], donde se emplea *Python* como lenguaje de programación para tareas como por ejemplo el desarrollo de herramientas y prototipos o la automatización de tareas.
- **Desarrollo rápido de prototipos**: Estos lenguajes permiten construir más fácilmente modelos de las aplicaciones que se van a desarrollar, que permitan comprobar ciertos aspectos del diseño, la validez de alguna idea o bien un aspecto general de cómo el sistema será finalmente, gracias a sus características de desarrollo interactivo y la integración del tiempo de desarrollo y ejecución.

A pesar de las características descritas, estos lenguajes no están tan extendidos como los "estáticos" para el desarrollo de aplicaciones, aunque su uso pueda reportar ventajas dadas las características de la propia aplicación a desarrollar o de su entorno. Entre los motivos que pueden originar esto, uno de los más significativos es la carencia de rendimiento en tiempo de ejecución de las aplicaciones finales, motivada principalmente por la necesidad de realizar durante la ejecución multitud de operaciones (como inferencia de tipos), que no son necesarias en un lenguaje "estático". Esto es un factor que puede tener un gran peso en el desarrollo y que haga que otras consideraciones, como la productividad (estimada como el tiempo empleado en el desarrollo de la aplicación), permanezcan en un segundo plano a la hora de elegir un lenguaje.

Entre las características más importantes de un lenguaje dinámico podemos encontrar las siguientes:

5.1.1 Sistema de Tipos Dinámico

En aras a mantener su flexibilidad en tiempo de ejecución, los lenguajes dinámicos, una vez compilados, hacen la comprobación e inferencia de tipos en tiempo de ejecución, permitiéndose pues que un objeto pueda modificar su tipo en función del camino seguido por la ejecución del programa. Esto puede ocasionar la detección de ciertos errores de programación en tiempo de ejecución, ya que no serán detectados durante la compilación del lenguaje.

No obstante, una forma de disminuir el número de posibles errores de este tipo que se puedan producir es emplear programas de *test* automatizados, que ayuden a su detección prematura, como *PyUnit* [PyUnit04], *Junit* [Junit06] o *Nunit* [Nunit05] o bien analizadores estáticos de programas, como *PyCheker* [Laird02].

5.1.2 Capacidad para Examinar y Cambiar su Estructura y/o Comportamiento

Esta característica es precisamente la que dota a los lenguajes dinámicos de un alto grado de flexibilidad. Mediante la misma, un programa podría cambiar el conjunto de métodos y/o atributos de cualquiera de sus tipos o instancias o la semántica de un conjunto de primitivas del sistema, reflejándose el resultado de los cambios en la propia ejecución del programa inmediatamente. La reflexión es el mecanismo utilizado para hacer este tipo de operaciones, y el nivel de la misma implementado por el lenguaje [Ortin01] determinará que tipo de operaciones se podrán realizar. Los lenguajes dinámicos suelen ofrecer capacidades de introspección y reflexión estructural. Gracias a ellas, un lenguaje dinámico es un medio más adecuado para crear *software* adaptativo y adaptable [Gonzalez04].

- Un *software* adaptativo [Álvarez99] es capaz de monitorizar e interpretar las actividades que el usuario está realizando y cambiar su *interface* para acomodarse mejor al perfil del usuario que está empleándolo en este momento. De esta forma, un *software* adaptativo podría cambiar para ajustarse mejor al usuario, a sus conocimientos o bien simplemente a sus preferencias personales en diferentes áreas. En conclusión, en este caso es el *software* el que trata de adaptarse al usuario y no el usuario el que trata de adaptarse al *software*.
- Un *software* adaptable es aquél que permite al usuario controlar los cambios que se le pueden hacer al *software* para ejecutar la personalización de la que hablábamos anteriormente, ofreciendo una serie de opciones posibles que se podrían modificar y toda la ayuda o guía necesaria para efectuar dichas modificaciones, así como la posibilidad de crear nueva funcionalidad (incorporar código nuevo a la aplicación). Un *software* capaz de mutar en tiempo de ejecución como el descrito podrá ser implementado por tanto de forma más sencilla empleando un lenguaje que también permita cambios en tiempo de ejecución.

5.1.3 Integración de Diseño y Ejecución

En un lenguaje verdaderamente dinámico la "barrera" existente entre tiempo de desarrollo y tiempo de ejecución es muy delgada. Si un programa puede efectuar cambios a su estructura o a su semántica, y esos cambios se reflejan inmediatamente en el propio programa, cualquier cambio realizado mientras el programa esté en funcionamiento tendrá un impacto directo en la propia ejecución del mismo, por lo que ya no se puede establecer una diferenciación clara entre ambas fases.

5.1.4 Manipulación de Diferentes Entidades como Objetos de Primer Orden

Los lenguajes dinámicos convierten muchas de las entidades que usan (módulos, metaclases, tipos, clases y objetos) así como los elementos que las forman (métodos y atributos) en objetos de primer orden, permitiendo que estos elementos puedan ser manipulados de la misma forma que un objeto de usuario, y haciendo por tanto más accesibles las entidades sobre las que se permiten hacer todos los cambios deseados. Esto hace que el código pueda factorizarse de un modo superior respecto a lenguajes que no ofrezcan estas características y que sean empleados en *frameworks* de aplicaciones.

5.1.5 Generación Dinámica de Código

Un lenguaje dinámico hace más sencilla la labor de generar dinámicamente código dentro de un programa (programación generativa), es decir, la capacidad de que un programa cualquiera pueda generar otro o bien partes del mismo. Ésta es una característica que poseen muchos lenguajes dinámicos y se apoya fundamentalmente en sus capacidades de reflexión estructural, que permiten modificar la estructura y comportamiento de cualquier clase, y por tanto podrán generar dinámicamente ese nuevo comportamiento en función de las decisiones motivadas por el estado actual de la aplicación si así lo desea. No obstante, muchos lenguajes dinámicos no ofrecen sólo la ampliación de clases existentes mencionada, sino también la creación de código que puede ser guardado en un fichero nuevo y cargado dinámicamente, de manera que pase a formar parte de la aplicación.

Ejemplos operativos de la generación dinámica de código para automatizar la creación de funcionalidades dentro de un programa, a partir de información contenida en ficheros de configuración o bases de datos, son el servidor de aplicaciones *web Zope* [Zope06] o en el *framework* para la creación, despliegue y mantenimiento ágil de aplicaciones *web Ruby on Rails* [Thomas05].

5.1.6 Carácter de Propósito general

Aunque en ocasiones los lenguajes dinámicos aparezcan identificados como lenguajes de *scripting*, es preciso hacer una separación entre ellos basándose

principalmente en su ámbito de aplicación. Si bien los lenguajes de *scripting* se usan principalmente para la unión entre componentes de una aplicación, como veremos en la sección siguiente, los lenguajes dinámicos poseen más características que les permiten ser un medio para desarrollar aplicaciones finales. Características como el acceso a bases de datos, la orientación a objetos y la creación de *GUI* (*Graphical User Interfaces*) hacen que estos lenguajes sean un medio válido para construir aplicaciones completas. Al comienzo del capítulo hemos visto ejemplos de estas aplicaciones.

5.2 LENGUAJES DE SCRIPTING

Los lenguajes de *scripting* [Ousterhout98] [Prechelt02] (también denominados lenguajes de *script*) son lenguajes de programación creados fundamentalmente para unir componentes desarrollados en lenguajes con más funcionalidades (lenguajes en los que se desarrollarían las aplicaciones finales), y llevar a cabo operaciones más complejas mediante su interacción (encadenamiento y ejecución de una determinada serie de tareas o trabajos de forma controlada y en un orden determinado), controlada por estos lenguajes. En sus inicios estos lenguajes eran llamados lenguajes *batch* o lenguajes de control de trabajos. Un lenguaje de este tipo es normalmente interpretado, aunque existen también casos de lenguajes compilados.

La evolución de este tipo de lenguajes con el tiempo ha llevado a la creación de lenguajes mucho más sofisticados, que permiten la escritura de programas más potentes y versátiles. A pesar de que actualmente estos programas adquieren una funcionalidad mucho más completa que la simple coordinación y lanzamiento automático y ordenado de trabajos, normalmente son llamados también *scripts* en referencia al tipo de lenguaje con los que se crean. Se pueden encontrar ejemplos de lenguajes de *scripting* a muchos niveles, todos ellos adaptados a un propósito o propósitos particulares: En el sistema operativo [WinScript06], como parte de videojuegos, para el desarrollo de páginas *Web*, documentos, etc.

5.2.1 Descripción

Como hemos visto, los *scripts* actúan estableciendo una unión entre diferentes componentes preexistentes ("actores"), para que éstos puedan realizar una tarea nueva ("guión" o "*script*") empleando dichos componentes de una determinada forma. Estos lenguajes además comparten una serie de propiedades comunes [Ousterhout98]:

- Favorecen un desarrollo más rápido del programa, dejando en un segundo plano su eficiencia en tiempo de ejecución.
- Son implementados a menudo por intérpretes en vez de compiladores, ejecutados cada vez que se invocan.
- No poseen un sistema de tipos estático, permitiendo una gran flexibilidad.
- Pueden comunicarse fácilmente con componentes escritos en otros lenguajes.

Debido a la naturaleza de estos lenguajes, es comprensible que muchos de ellos

se crearan específicamente para abordar tareas específicas, lo que condiciona sus capacidades y posibilidades, al estar orientados a un propósito concreto. Tareas como ayuda a la administración de sistemas, creación de interfaces de usuario gráficas o ejecutar una serie de comandos siguiendo un orden y periodicidad concreta automáticamente [Gite06], que de otra forma deberían ejecutarse introduciéndolos "a mano" por parte del usuario, son tareas comunes abordadas con estos lenguajes.

Es posible encontrar lenguajes de *script* que sean capaces de escribir programas más generales, no tan vinculados a un dominio específico. En algunos proyectos de gran tamaño un lenguaje de *script* se usa de forma conjunta con un lenguaje de programación a bajo nivel, ocupándose el primero de tareas que permiten sacar provecho de sus características. También existen aquellos diseñados para ser interactivos, presentando un *prompt* al usuario en el que dicho usuario deberá introducir individualmente comandos que representan operaciones a alto nivel, como ocurre con el *shell* de *Linux* llamado *bash* [Cooper06].

Estos lenguajes pueden presentar características avanzadas, como el manejo automático de memoria o chequeos de rango automáticos, labores de las que el programador quedará liberado, pudiéndose producir potencialmente un menor número de errores derivados. Estas características (entre otras), junto a su naturaleza interpretada, son las que contribuyen a la menor eficiencia de este tipo de lenguajes, al no permitir una optimización del programa (tanto en velocidad de ejecución como en consumo de memoria) tan avanzada como en lenguajes que no las posean. Por ello, los lenguajes de *script* se usan en aquellos entornos en los que las ventajas de flexibilidad que ofrecen los lenguajes de *script* compensan su falta de eficiencia.

Actualmente, con la aparición de nuevos lenguajes de *scripting* con más funcionalidades, la diferencia entre éstos y los lenguajes de programación "de sistemas" es cada vez menos patente, existiendo lenguajes de *scripting* en los que es posible hacer prácticamente lo mismo que en uno de propósito general.

5.2.2 Tipos de Lenguajes de Script

Haremos ahora una posible división de las diferentes clases de lenguajes de *scripting* existentes teniendo en cuenta su función como criterio principal, basándose en lo expuesto anteriormente:

5.2.2.1 LENGUAJES DE PROCESAMIENTO DE TEXTOS:

Una de las acciones más antiguas que se podían hacer con lenguajes de *script* es el procesamiento de información basada en texto plano. Algunos lenguajes de este estilo, como el *awk* de *Unix* o el *Perl* [OReilly06], fueron originalmente creados para ayudar a los administradores del sistema a automatizar tareas, leyendo la información de archivos de configuración de *Unix* (en texto) o bien de archivos de *Log*. En este apartado puede destacarse la presencia del lenguaje *Perl*, originalmente pensado como lenguaje para la generación de informes, pero que ahora ha evolucionado en un lenguaje capaz de generar aplicaciones completas. De la misma forma se puede citar al lenguaje *PHP*, creado inicialmente como un lenguaje especializado para la creación dinámica de contenidos *Web*, pero usado actualmente también para tareas de administración general. Ambos lenguajes serán descritos posteriormente. Dentro de esta categoría podemos citar también a *XSLT*, un procesador de plantillas que, combinando estas plantillas con los datos existentes en un documento *XML*, permite la transformación de un documento *XML* en otro documento *XML* que pueda tener, por ejemplo, otro formato distinto al original

[W3CXSLT99].

5.2.2.2 LENGUAJES DE CONTROL DE TRABAJOS Y *SHELLS*:

Como ya se ha dicho, una de las principales tareas de un lenguaje de *script* era el control de trabajos, permitiendo iniciar y controlar el comportamiento de programas del sistema. Algunos de los intérpretes de estos lenguajes funcionan también como interfaces en línea de comandos, adquiriendo la forma de *shells* o línea de comandos del estilo a lo que podemos encontrar en sistemas operativos de la familia *Unix/Linux* o *Windows*. No obstante, existen otros que no requieren este tipo de interfaces de línea de comandos, como *AppleScript*. Algunos lenguajes de este tipo son: *bash* [Cooper06], *csh* [Hawaii01], *JCL* [Computer06].

5.2.2.3 LENGUAJES VINCULADOS A UNA APLICACIÓN:

Muchas aplicaciones de gran tamaño incluyen un lenguaje de *script* específicamente creado a medida para la tarea o tareas a desarrollar por dicha aplicación y/o los usuarios de la misma. Por ejemplo, hoy en día muchos videojuegos incluyen lenguajes de *script* para expresar las acciones de aquellos personajes no controlados por el jugador o bien para regir el comportamiento de ciertos objetos del entorno de juego.

Algunos de estos lenguajes son similares o surgen a partir de lenguajes "estándar" de propósito general, como el caso de *QuakeC* [QuakeC06].

5.2.2.4 LENGUAJES DE PROPÓSITO GENERAL:

Algunos lenguajes que inicialmente fueron concebidos como de *script* han evolucionado en lenguajes de programación de propósito más amplio, hasta el punto de ser comparables a lenguajes que tradicionalmente han tenido un propósito general. Normalmente lenguajes, que poseen una determinada serie de características (interpretados, con gestión automática de memoria y dinámicos), son incluidos dentro de esta categoría y denominados en ocasiones lenguajes de *scripting*, aunque no sea del todo preciso referirse a los mismos de esta forma dado su propósito. Podemos incluir a *Python* o *Ruby* dentro de esta categoría, pero nosotros no los trataremos como lenguajes de *scripting* debido a que incluyen multitud de funcionalidades que les diferencian de éstos.

5.2.2.5 LENGUAJES "INCRUSTADOS":

Un número determinado de lenguajes se han diseñado con el propósito de reemplazar lenguajes que son diseñados para una aplicación concreta, de forma que se puedan "incrustar" en programas de aplicación. El programador creará el programa en *C* u otro lenguaje equivalente e incluirá puntos de unión donde el lenguaje de *scripting* podrá tomar el control de la aplicación. Estos lenguajes sirven para las mismas tareas que los lenguajes vinculados a una aplicación concreta, pero permitiendo ser usados entre aplicaciones. Algunos lenguajes que pueden funcionar de este modo son *ECMAScript* (incluyendo bajo esta denominación a *ActionScript*, *DMDScript*, *JavaScript*, *JScript*) y *Tcl* (aunque este lenguaje también puede trabajar de forma no incrustada

[Tcl06]).

5.2.3 PHP

PHP [PHP06] es un lenguaje de programación reflectivo y de código abierto (con licencia de tipo *BSD*) usado mayoritariamente para el desarrollo de aplicaciones del lado del servidor y *Web* de contenido dinámico, aunque también puede dársele otros usos. Existen aplicaciones desarrolladas con *PHP* que han adquirido una considerable fama, como *phpBB* [PHPBB06] o *MediaWiki* [MediaWiki06]. El modelo de implementación de *PHP* es una alternativa a la tecnología *ASP .NET* de *Microsoft* u otros sistemas como *JSP* de *Sun* o *ColdFusion* de *Macromedia*, entre otros.

Una de las principales características de *PHP* es su gran capacidad para integrarse fácilmente con un determinado conjunto de tecnologías, que permiten su empleo para un gran número de funciones. De esta forma, se permite la interacción con un alto número de sistemas de gestión de bases de datos relacionales, como *MySQL* [MySQL06] u *Oracle* [Oracle06], existiendo implementaciones sobre un gran número de sistemas operativos como *Linux* o *Windows,* y contando además con la posibilidad de colaborar con la mayoría de los servidores *Web* más usados en el mercado, como *Apache* [Apache06] o *IIS* [IIS06]. También es destacable la existencia de un gran número de librerías que extienden la funcionalidad del mismo mucho más allá de las capacidades poseídas por el núcleo de su implementación, permitiendo su uso de servidores de *FTP*, servidores *LDAP* y otras aplicaciones.

PHP es un lenguaje que posee resolución dinámica de tipos, no existiendo pues reglas estrictas en cuanto a declaración de variables, por lo que en un momento dado una variable puede valores de tipos diferentes en función del camino de ejecución seguido por el programa. Por tanto, este lenguaje no necesita que se declare explícitamente el tipo de un dato, sino que éste es inferido en tiempo de ejecución teniendo en cuenta el contexto en el que se usa dicho dato. *PHP* infiere tipos comunes como *boolean*, *integer*, números en punto flotante y *string*. Adicionalmente, *PHP* permite obtener fácilmente el tipo de un dato cualquiera en un momento dado mediante la función integrada `gettype()` y modificarlo usando `settype()`. También se permiten operaciones de *casting*, efectuando una copia del dato original convertida al tipo especificado [Software06]. Esta concepción de los tipos de *PHP* también se plasma en el tratamiento realizado por el lenguaje de los *arrays*. Los *arrays* son heterogéneos, de manera que un *array* puede contener objetos de diferentes tipos.

5.2.3.1 PROGRAMACIÓN ORIENTADA A OBJETOS CON *PHP*

Hasta la versión 3, *PHP* no tenía capacidades de orientación a objetos. A partir de la mencionada versión se implementaron una serie de capacidades básicas, que fueron potenciadas ligeramente en la versión 4. Es a partir de la versión 5 [Suraski06] cuando las capacidades de orientación a objetos de *PHP* se remodelaron, dándoles un carácter más robusto y completo y mejorando enormemente las versiones precedentes.

En esta última versión, la manera que *PHP* tiene de tratar objetos ha sido completamente rediseñada para incluir nuevas características y mejorar el rendimiento. Siguiendo siempre un modelo de orientación a objetos basado a clases clásico, todo objeto es accedido usando un manejador asociado y no su valor, lo que permite ahorrarse operaciones de copia que penalizarían el rendimiento cuando se pasa como parámetro o es asignado. Además se introduce la noción de miembros protegidos y

privados, así como clases y métodos abstractos, de la misma forma que se implementan en otros lenguajes, como *C++*.

Otras características interesantes introducidas en esta versión 5 son los *interfaces* (permitiendo a las clases implementar un número cualquiera de ellos), la clonación controlada de objetos (permitiendo al programador definir cómo se va a duplicar un objeto) o el uso de constructores y destructores. Por último, también es destacable la presencia de excepciones. Se pueden encontrar más características en el manual oficial de *PHP* 5 [Achour06].

5.2.3.2 REFLEXIÓN EN *PHP*

Para ilustrar más en profundidad las capacidades dinámicas poseídas por *PHP*, se describirán a continuación sus capacidades de reflexión. *PHP* 5 incorpora una *API* de reflexión que permite al programador diversas operaciones, como hacer ingeniera inversa de clases, interfaces, funciones, métodos y excepciones, permitir obtener los comentarios de funciones, métodos y clases y otro tipo de operaciones útiles [Achour06b]. El *API* de reflexión de *PHP* está formado por las siguientes clases:

```
<?php
class Reflection { }
interface Reflector { }
class ReflectionException extends Exception { }
class ReflectionFunction implements Reflector { }
class ReflectionParameter implements Reflector { }
class ReflectionMethod extends ReflectionFunction { }
class ReflectionClass implements Reflector { }
class ReflectionObject extends ReflectionClass { }
class ReflectionProperty implements Reflector { }
class ReflectionExtension implements Reflector { }
?>
```

Como puede verse, el *API* de reflexión de *PHP* posee una clase para trabajar con cada uno de los elementos principales de un programa cualquiera (clases, métodos, etc.). Cada una de estas clases permitirá trabajar adecuadamente su elemento asociado, aportando un determinado conjunto de operaciones sobre ellos. Para ver un ejemplo de qué tipo de operaciones se soportan sobre los elementos del lenguaje, a continuación mostramos parte de la interfaz de la clase `ReflectionFunction`:

```
<?php
class ReflectionFunction implements Reflector
{
   public string getName()
   public string getFileName()
   public int getStartLine()
   public int getEndLine()
   public string getDocComment()
   public array getStaticVariables()
   public mixed invoke(mixed* args)
   public mixed invokeArgs(array args)
   public bool returnsReference()
   public ReflectionParameter[] getParameters()
   public int getNumberOfParameters()
   public int getNumberOfRequiredParameters()
}
?>
```

Vemos cómo se incorporan operaciones que permiten saber todo tipo de detalles sobre las funciones, como su nombre, comentarios, tipo y número de parámetros, etc. e

incluso la invocación de la misma. Existe un soporte similar para los otros elementos del lenguaje mencionados.

En resumen, podemos ver como *PHP* ofrece soporte para hacer reflexión de sólo lectura (introspección) sobre los elementos de su modelo de objetos basado en clases tradicional, permitiendo obtener mucha información en tiempo de ejecución. Mediante el uso del *API* de reflexión de *PHP* también se puede lograr la carga dinámica de clases, aunque no de una forma directa, sino mediante el empleo de una función específica para ello del *API*.

5.2.4 Ecma/J/JavaScript

ECMAScript es un lenguaje de *scripting* estandarizado en la especificación *ECMA-262* [ECMA26299] por *Ecma International* [ECMA06]. Este lenguaje está muy extendido para proporcionar un soporte a la creación de páginas *Web,* usándose para esta función variantes como *Javascript* o *Jscript*. Estos dos últimos son implementaciones del mismo creadas por diversos fabricantes con extensiones propias, surgidas a lo largo de la historia. En el año 1995, la empresa *Sun Microsystems,* junto con la corporación de comunicaciones *Netscape,* introdujeron por primera vez *JavaScript*. Un año después, *Netscape* lanzo al mercado el navegador *Netscape 2.0*, con un soporte directo para este lenguaje. Debido al gran éxito cosechado por este navegador y *Javascript* como medio adecuado para mejorar la creación de paginas *Web*, *Microsoft* creó un lenguaje similar llamado *Jscript*, que no era del todo compatible con el primero, incluyéndolo por primera vez en la versión 3.0 de su navegador *Internet Explorer*. *Netscape* envió la especificación de *JavaScript* a *Ecma International* con el objetivo de lograr su estandarización, estándar que se creó bajo la denominación *ECMA-262* en el año 97. *ECMAScript* es por tanto el nombre del lenguaje estandarizado, mientras que tanto *Javascript* como *Jscript* son implementaciones que intentan ser completamente compatibles con éste estándar, pero que proporcionan funcionalidades adicionales no contempladas en la especificación *ECMA*.

Debido pues al alto grado de similitud entre estos tres lenguajes, tomaremos uno de ellos como modelo para exponer las características generales de los demás.

5.2.4.1 JAVASCRIPT

JavaScript es un lenguaje de programación basado en objetos, orientado a prototipos, interpretado y que no usa el modelo "tradicional" basado en clases de lenguajes como *C++*. La última versión conocida en el 2005 es la versión 1.5, correspondiente al estándar *ECMA-262 v3.* Como ya se mencionó anteriormente, su uso más típico es la creación de paginas *Web* dinámicas, que permitan hacer operaciones de mayor complejidad y utilidad, pero también suele ser usado para acceder mediante a *scripting* a objetos que están integrados en otras aplicaciones. Posee una sintaxis similar a la de *C*, pero tiene más puntos en común con el lenguaje *Self* (descrito también en este capítulo). A pesar de su nombre, no existe ninguna relación real entre *Java* y *Javascript* aparte de la sintaxis derivada de *C* de ambos. Una de las principales ventajas del uso de *Javascript* es que, a diferencia de otros lenguajes de *scripting* similares como *VBScript* [MSDNVB06], es soportado por multitud de navegadores (no sólo *Internet Explorer*).

Una característica peculiar de este lenguaje es que no posee herramientas propias para hacer entrada/salida. A diferencia de *C*, que necesita librerías para esta tarea, el motor de *JavaScript* necesita un programa externo para ello en el que se integra, como un navegador *Web*. Por ello, actualmente uno de los usos más comunes de la tecnología *Javascript* es escribir funciones que se integran dentro de las paginas *HTML*, y que

pueden interactuar con el *DOM* de la pagina (*Document Object Model:* Una representación jerárquica y ordenada en forma de objetos del contenido de la pagina) para hacer tareas que no son posibles usando sólo *HTML* estático: Abrir ventanas, comprobar la validez de los datos, etc. El principal problema existente con *Javascript* es que algunos navegadores no siguen el estándar de *DOM* especificado por el *W3C* [W3DOM06], con lo que es posible que diferentes navegadores no ofrezcan los mismos objetos y/o métodos a un *script* dado, obligando pues al programador a hacer *scripts* personalizados para cada uno de ellos y eliminando de esta forma toda posibilidad de hacer estos *scripts* fácilmente portables. Otras aplicaciones que no son navegadores, como *Adobe Acrobat* [Adobe06] también soportan *Javascript* en archivos *PDF*. Estas aplicaciones ofrecen un *DOM* distinto, adaptado a las características propias del entorno al lenguaje, que no cambia. En lo referente a *Jscript*, la ultima evolución del mismo es el lenguaje *Jscript .NET* [MSDNJS06], parte de la tecnología *.NET* de *Microsoft* y que aporta a *Jscript* más características de programación orientada a objetos.

FUNCIONAMIENTO GENERAL

[SunJScript06] Si nos ceñimos al uso más común del lenguaje *Javascript* (integrarse en páginas *HTML*), las propiedades de un objeto en *Javascript* están basadas en el contenido del documento *HTML* con el que el código está relacionado, es decir, que las propiedades reflejan el contenido de la pagina *HTML*. Dado que los navegadores generalmente empiezan a procesar la página de arriba a abajo, mostrando y procesando todo el código con el que se van encontrando en ese orden de lectura, *Javascript* sólo refleja propiedades y elementos que se encuentren en el *HTML* que ya ha sido procesado. Por ejemplo, si se define el siguiente formulario con dos cajas de texto para incluir información:

```
<FORM NAME="statform">
<INPUT TYPE = "text" name = "userName" size = 20>
<INPUT TYPE = "text" name = "Age" size = 3>
```

Estos elementos son reflejados como objetos del lenguaje que se pueden usar después de que el formulario ha sido definido, pudiendo acceder en ese momento a las propiedades `document.statform.userName` y `document.statform.Age`. Un *script* que simplemente muestre el valor de las propiedades sería:

```
 <SCRIPT>
document.write(document.statform.userName.value)
document.write(document.statform.Age.value)
</SCRIPT>
```

Este *script* por tanto no funcionaria si se intenta ejecutar antes de la propia definición del formulario en la página. De la misma forma no es posible cambiar un elemento de valor si el navegador ya lo ha mostrado. Si por ejemplo tenemos que una pagina tiene su titulo definido de esta forma:

```
<TITLE>Mi pagina Javascript</TITLE>
```

Este valor se puede obtener en el objeto *JavaScript* `document.title`. No obstante, una vez que el navegador lo ha mostrado, no se puede cambiar el valor de esta propiedad, ya que de hacerlo no tendría efectos prácticos sobre la visualización de la misma, al haber sido ya mostrada esta parte (aunque tampoco da error). La única forma

que este lenguaje tiene para hacer este tipo de operaciones de cambio dinámico de valor con los elementos de un formulario es mediante el uso de eventos. Por ejemplo, el siguiente código añade el texto "Click" al campo de texto creado en el formulario cada vez que el usuario hace *click* en un botón añadido al efecto:

```
<FORM NAME="demoForm">
<INPUT TYPE="text" NAME="mytext" SIZE="40" VALUE="Valor inicial">
<P><INPUT TYPE="button" VALUE="Haz Click para actualizar el TextField"
   onClick="document.demoForm.mytext.value += '...Click' ">
</FORM>
```

Por tanto, todo cambio en el valor de un elemento que quiera hacerse visible al usuario pasa por hacer una recarga de la página irremediablemente, aunque actualmente existen medios para aumentar el rendimiento de estas recargas y optimizar este proceso.

CARACTERÍSTICAS DE FLEXIBILIDAD Y REFLEXIÓN

El sistema de tipos poseído por *Javascript* es dinámico, de manera que todo tipo es inferido en tiempo de ejecución en función del contexto en el que la aplicación se ejecuta. Básicamente, el lenguaje entiende tres tipos de atributos básicos: números, cadenas de caracteres y booleanos. Una variable dada podrá tomar cualquiera de estos valores a lo largo de la ejecución de un programa y será durante la misma cuando el sistema analice el valor de dicha variable y la interprete en función de los datos que posea [Devarticles06].

Por otra parte, el modelo orientado a prototipos empleado por el lenguaje [Predescu06] hace que la creación de objetos en el mismo difiera de la clásica instanciación del modelo de clases. Por ejemplo, si deseamos crear un objeto cualquiera debemos definir una función estándar, a la que se llamará función constructor, y que dará un valor inicial al estado del objeto recién creado. Un ejemplo de este tipo de funciones es:

```
function Miclase(){
        this.atributo1 = 1;
        this.atributo2 = "2";
}
```

Este constructor es el que realmente especificará cómo será el prototipo de todos los objetos de tipo "MiClase". Para crear entonces una instancia de esta clase simplemente habrá que hacer:

```
var obj = new MiClase();
```

El operador `new` creará un nuevo objeto *Javascript* e invocará la función asociada con el nombre "MiClase", pasando un argumento invisible *this*, que será el objeto recién creado. Dentro de esta función el código usa la propiedad *this* para referirse precisamente a la instancia. Si por ejemplo queremos añadir un atributo nuevo llamado "propiedad1" a los objetos, simplemente tenemos que modificar el constructor así:

```
function MiClase() {
        this.atributo1 = 1;
        this.atributo2 = "2";
        this.propiedad1 = 1;
}
```

Vemos pues con este ejemplo como *Javascript* posee el modelo de objetos orientado a prototipos, donde se crea un objeto prototipo (a través de la función constructor) a partir del cual se pueden clonar todas las instancias del mismo que necesitemos (a través del operador `new`) y que luego podremos modificar según nuestras necesidades añadiendo atributos y métodos a cada uno de ellos (reflexión estructural). Para añadir métodos a los objetos simplemente se debe crear el método con la funcionalidad requerida, como por ejemplo:

```
function muestraMiClase() {
   var result = "Atributo 1: " + this.atributo1
                + " " + this.atributo2;
   pretty_print(result)
}
```

Y posteriormente añadírselo a cada objeto sobre el que queramos usarlo, mediante una instrucción como:

```
this.muestraMiClase = muestraMiClase;
```

Pasando a poder usar el método con el objeto a partir de entonces. En *Javascript* se pueden crear objetos (que pueden incluso estar inicialmente vacíos) y posteriormente ampliarlos todo lo necesario por programa, añadiéndoles métodos y propiedades (simplemente dándoles un valor para ese objeto [W3Schools06]) en tiempo de ejecución. *Javascript* no sólo soporta la modificación de un objeto particular, como hemos visto hasta ahora, sino que también permite añadir propiedades y métodos al objeto prototipo de las mismas, como se muestra en el siguiente ejemplo [Javascriptkit06]:

```
//Creamos la función constructor de círculo
function circulo(){
...
}

circulo.prototype.pi=3.1416
```

Esta técnica, que puede emplearse también para añadir métodos, hace que a partir de ese instante todos los objetos derivados del prototipo de `circulo` tengan la propiedad `pi`. El lenguaje también permite hacer lo mismo con los objetos preconstruidos que ofrece al programador para diversas tareas, pero sólo si éstos se crean mediante `new`. Mediante `removeAttribute` es posible eliminar los atributos de un objeto.

Por tanto, según hemos visto en el capítulo anterior, *Javascript* posee un modelo de herencia por delegación. En lo referente precisamente a la herencia, *Javascript* no tiene una palabra reservada para este fin, pero se puede lograr este comportamiento mediante dos técnicas [WebReference06]. La primera de ellas consiste en llamar a la función constructor del que queremos que sea el prototipo "padre" desde dentro de la función constructor del "hijo":

```
function superClase() {

  this.adios = superAdios;
  this.hola = superHola;
}

function subClase() {
  this.inheritFrom = superClase;
  this.inheritFrom();
```

```
  this.adios = subAdios;
}

function superHola() {
  return "Hola desde superClase";
}

function superAdios() {
  return "Adios desde superClase";
}

function subAdios() {
  return "Adios desde subClase";
}

function printResul() {
  var newClase = new subClase();
  alert(newClase.adios());
  alert(newClase.hola());
}
```

El método `adios` está redefinido en la subclase y es el que se llamará en el ejemplo. La segunda técnica [WebReference06] consiste en crear un objeto de la superclase y asignárselo al objeto `prototype` visto anteriormente. En el ejemplo anterior bastaría con eliminar las líneas que llaman al constructor en la función `subClase` e incluir ésta:

```
subClass.prototype = new superClass;
```

Esta forma de lograr herencia es mejor que la anterior ya que, a diferencia de ésta, soporta herencia dinámica, permitiendo definir métodos y propiedades en la superclase después de que se ejecute la función constructor y que las subclases los adquieran automáticamente.

[Noras04] Además de las capacidades de reflexión estructural descritas, *Javascript* posee la capacidad de introspección sobre cualquier instancia. Por ejemplo, es posible iterar y examinar todos los miembros de un objeto, o acceder a todos los objetos definidos (usando el operador `[]` con el nombre del objeto sobre el ámbito global, al que se puede acceder mediante `this[…]`) usando la instrucción `for`.

Por último, cabe destacar que la especial estructura y funcionamiento de *Javascript* [MozillaJavaS06] le dota de capacidades dinámicas avanzadas, incluyéndose características como la construcción en tiempo de ejecución de objetos, *closures*, listas de parámetros variables, creación dinámica de *scripts* (mediante el uso de `eval`), la introspección de objetos ya vista y la capacidad de descompilar el cuerpo de funciones y convertirlas de nuevo en código fuente.

5.2.5 Perl

Perl [Wall00] es un lenguaje de programación de propósito general originalmente desarrollado para la manipulación de texto, pero cuya evolución le ha llevado a su aplicación en otro tipo de tareas diversas, como la administración de sistemas operativos, desarrollo *Web*, programación de redes, etc., aunque una de sus aplicaciones más comunes es la creación de scripts *CGI*.

La estructura general de *Perl* [OReilly06] recuerda vagamente a *C*. *Perl* es un lenguaje de programación procedural, con variables, expresiones, bloques, etc.

[Emerson06]. Los programas desarrollados en *Perl* eran originalmente interpretados, aunque implementaciones modernas emplean una máquina virtual para esta tarea. En *Perl*, todos los nombres de variables están precedidos por un determinado símbolo (`$`, `@`, `%`), que indica el tipo de variable al que se refiere (variable, *array*, tabla, etc.) en función de cual se use. Esto permite añadir las variables directamente a cadenas de caracteres. *Perl* posee también un número muy elevado de módulos que le proporcionan acceso a un gran abanico de funcionalidades que se puedan necesitar.

A partir de la versión 5, *Perl* cuenta con soporte para estructuras de datos más complejas, como funciones de primer orden, y un modelo computacional orientado a objetos basado en clases similar al de lenguajes como *C++* (clases, atributos, métodos, etc.), así como módulos que permitan la fácil redistribución de su código. Todas las versiones de este lenguaje tienen un sistema de tipos dinámico que hace inferencia de tipos automática (las conversiones ilegales de tipo son tratadas como errores fatales). Además el intérprete lleva un seguimiento preciso de cada objeto en el programa, proporcionando servicios de recolección de basura y manejo automático de memoria.

Perl es un lenguaje que tiene muchos defensores y también detractores. A favor del mismo podemos citar que es un lenguaje muy expresivo y fácil de usar, su potente soporte para el tratamiento de listas y textos en general y el manejo automático de memoria. En cuanto a sus principales problemas, cabe citar que el código desarrollado con el mismo puede hacerse difícilmente legible, dada su particular estructura, achacándole ser demasiado complejo y compacto, y que no cuenta con una especificación estándar publicada como en el caso de otros lenguajes.

Perl no introduce ninguna sintaxis especial para definir objetos, clases o métodos, sino que reutiliza las estructuras que ya posee para implementar estos conceptos. El modelo de objetos de *Perl* [Wall00b] se basa en tres principios [Cpan06]:

- Un objeto es una referencia que conoce a que clase pertenece.
- Una clase es un paquete que posee métodos que pueden trabajar con referencias a objetos.
- Un método es una subrutina que espera recibir una referencia a un objeto (o a un paquete para métodos de clase) como primer argumento.

En este lenguaje tampoco existe una sintaxis especial para los constructores, sino que éstos son simplemente subrutinas que devuelven una referencia a un objeto, que se asocia a una clase mediante una operación del lenguaje denominada "*blessing*":

```
package Criatura;
   sub new {
    my $self = {};
    bless $self;
    return $self;
}
```

Los constructores pueden tener pues cualquier nombre (`new` se ha colocado sólo por similitud con otros lenguajes). Esta operación devuelve una referencia a un objeto a la que se le ha asociado el paquete `Criatura` como "clase" de la misma.

Un constructor puede también cambiar la clase a la que está asociada un objeto (volver a hacer la operación *bless* sobre otra clase), permitiendo así el cambio dinámico de clase de un objeto. Un objeto sólo puede pertenecer a una clase en un momento dado. A la hora de definir una clase, cuando se crea el `package` que identifica su nombre se deben simplemente incorporar definiciones de métodos dentro del mismo para

formarla. Dentro de una clase existe un *array* especial llamado `@ISA`, que determina los ancestros o padres de esta clase, y que se usa para buscar métodos en el caso de que se llame a uno desde un objeto cuya clase no lo tenga definido. Ésta es la forma mediante la cual *Perl* implementa la herencia (simple y múltiple), y cada elemento de ese *array* (que será el nombre de otro paquete o clase) se usa para la búsqueda de métodos según el orden en el que se han añadido al mismo (implementándose pues una estrategia de herencia por delegación).

Todas las clases heredan de la clase `UNIVERSAL`, padre de todas las clases. Si en la búsqueda de métodos se encontrase un método en alguna clase padre del objeto sobre el que se invoca, este método se cachearía en la clase actual para ganar en eficiencia. Este lenguaje posee herencia dinámica, puesto que permite cambiar en tiempo de ejecución el contenido del *array* `@ISA`, lo que hace que esta caché se borre en este caso (lo mismo ocurre si se definen nuevos métodos). En caso de que la búsqueda de métodos no sea satisfactoria, se llama por defecto al método `AUTOLOAD()` si es posible.

El lenguaje *Perl* sólo soporta herencia para métodos, los atributos se dejan dentro normalmente de cada una de clases, como variables de instancia, donde puedan tener valores propios a cada una de ellas. No obstante, *Perl* también soporta variables de clase, que serán compartidas por todas las instancias de la misma. Los objetos tienen una tabla *hash* propia donde guardan estas variables de clases, que serán usadas por todas las clases que trabajen con los mismos. Dado el soporte de este lenguaje para añadir variables de instancia a los objetos de forma dinámica, en ocasiones se llega a añadir el nombre de la clase que usa esa variable al nombre de la variable en sí para evitar errores por colisión de identificadores.

```
sub f {
    my $self = shift;
    $self->{ __PACKAGE__ . ".contador"}++;
}
```

En lo referente a métodos, no existe ninguna sintaxis especial para su definición (sí para su invocación). Un método siempre espera que el primer argumento que se le pase sea un objeto o paquete (sobre el que se invoca). Podemos decir entonces que este lenguaje tiene un soporte completo de reflexión estructural dinámica, al permitir añadir, borrar y modificar atributos de clases y objetos y modificar la jerarquía de clases en tiempo de ejecución. Aunque el modelo de objetos de *Perl* está basado en clases, existen paquetes adicionales que le dotan de un modelo basado en prototipos [Cpan06b] gracias a la flexibilidad poseída por el lenguaje.

Por último, haremos una descripción de las capacidades de introspección de *Perl*. Este lenguaje tiene una serie de características que permiten introspección [OReilly06b], obteniendo información acerca del contenido de la tabla de símbolos [OReilly06c]. Cada paquete tiene asociada una tabla de símbolos propia (llamada *stash* o "*symbol table hash*"). Estas tablas son accesibles como si fuesen un *array* asociativo normal. Para el paquete P, su tabla de símbolos será accesible usando `%P::`. Por ejemplo, para el paquete *main* se puede acceder a su tabla mediante `%main::` o `%::`. Todas las tablas de los paquetes son accesibles desde el paquete *main*, como aparece en la figura 5.1:

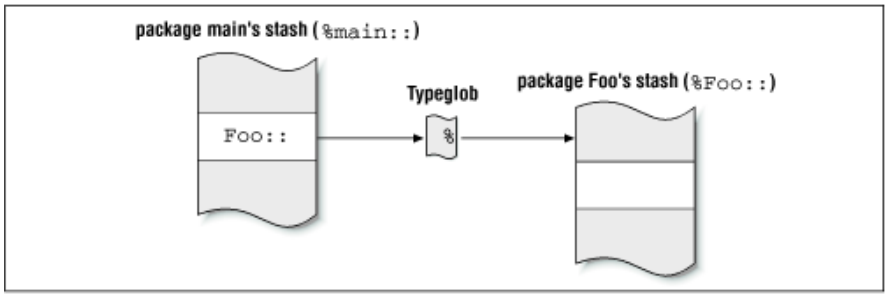

Figura 5.1: Acceso de tabla de símbolos de un paquete desde *main [OReilly06c]*

Para obtener todos los símbolos de un paquete podemos usar el siguiente código:

```
foreach $name (keys %main::) {
    print "$name, \n";
}
```

Cada nombre simbólico tiene asociada una entidad llamada *typeglob*, que puede apuntar a uno o varios valores (uno o más de un tipo concreto: escalar, *array*, *hash*, subrutina, manejador de fichero, etc.). Al no existir una forma directa de averiguar qué valores existen de entre todas estas posibilidades, un programador debe mirar manualmente las posibles opciones. El siguiente código [OReilly06c] ilustra este proceso:

```
package DUMPVAR;
sub dumpvar {
    my ($packageName) = @_;
    local (*alias);             # Un typeglob local
    # Accedemos al stash correspondiente al nombre del paquete
    *stash = *{"${packageName}::"};  # %stash es ahora la tabla de símbolos
    $, = " ";                        # Separador para imprimir
    # Iteramos la tabla de simbolos, que contiene valores glob indexados por los nombres
    # de los símbolos.
    while (($varName, $globValue) = each %stash) {
        print "$varName ============================ \n";
        *alias = $globValue;
        if (defined ($alias)) {
            print "\t \$$varName $alias \n";
        }
        if (defined (@alias)) {
            print "\t \@$varName @alias \n";
        }
        if (defined (%alias)) {
            print "\t \%$varName ",%alias," \n";
        }
     }
}
```

Un ejemplo de uso de este código y su salida sería:

```
package XX;
$x = 10;
@y = (1,3,4);
%z = (1,2,3,4, 5, 6);
$z = 300;
DUMPVAR::dumpvar("XX");

# Imprime
x ============================
         $x 10
y ============================
         @y 1 3 4
z ============================
```

```
$z 300
%z  1 2 3 4 5 6
```

Otro ejemplo de capacidades dinámicas es la función `eval`, que permite pasar una cadena de caracteres que tratará como código fuente, compilándolo y ejecutándolo en tiempo de ejecución, lo que permite ejecutar código a voluntad de forma condicional en medio de un programa, cargándolo incluso de fuentes externas, y por tanto la generación dinámica de código. Uno de los aspectos más potentes de esta capacidad es que el código ejecutado no es considerado parte de un programa separado, sino parte del código original, como se muestra en la figura 5.2:

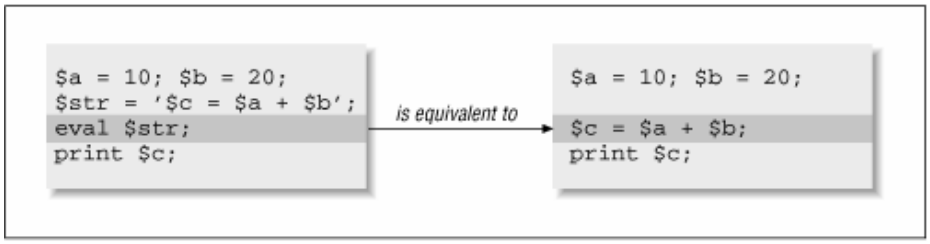

Figura 5.2: Uso del *eval* en *Perl* [OReilly06b]

Por este motivo, cualquier código pasado como parámetro a `eval` en forma de `string` puede usar cualquier variable o función definida en el programa hasta el momento, así como producir variables nuevas.

5.2.6 Tcl

Tcl [Zimmer98], cuyo nombre viene dado de "*Tool Command Language*", es un lenguaje funcional de *scripting* que es usado típicamente para hacer prototipos rápidos, *GUIs* (a través de su extensión *Tk*) y *tests*. También tiene su aplicación para la creación de *scripts CGI*. *Tcl* no proporciona soporte para la orientación a objetos, pero puede ser extendido para proporcionar nuevas características según sea necesario, y de hecho existen ya extensiones adecuadas para ello, como por ejemplo *XOTcl* [Zdun06], que será la que usemos para describirlas en esta sección. Entre las principales características del lenguaje podemos incluir:

- Todo es un comando, incluidas las estructuras del lenguaje. Se usa la notación polaca.
- Todo puede ser dinámicamente redefinido y sobrecargado.
- No posee tipos estáticos, cualquier variable puede contener cualquier tipo que se le asigne.
- Todos los tipos de datos pueden ser manipulados como `strings`, incluido el código fuente del programa.
- Las normas sintácticas son muy simples.
- Posee eventos, bien definidos por el usuario, temporizados o como respuesta a otros eventos.
- El código puede ser creado y modificado dinámicamente, al ser un lenguaje

interpretado. No obstante el rendimiento no se resiente excesivamente debido a su compilación a *bytecode*.

La sintaxis de *Tcl* es simple, consistiendo en un comando y una lista de argumentos para el mismo, separados todos por un espacio en blanco. Los comandos no son parte del lenguaje, sino de la librería:

```
commandName  argument1 argument2 ... argumentN
```

Cualquier argumento puede ser opcionalmente reemplazado por un comando entre corchetes, evaluándose primero este subcomando y sustituyéndose por el valor resultante de evaluarlo. Por tanto, a base de comandos y de reglas de sustitución se construyen básicamente todos los elementos de este lenguaje.

Describiremos ahora el modelo de objetos de *XOTcl* (que se basa en *Tcl*) para mostrar sus capacidades de reflexión y su modelo de objetos [Zdun06b] [XOTcl06], que en esta ampliación concreta está muy influenciado por el modelo de objetos de *CLOS*. *XOTcl* posee dos comandos principales `Object` y `Class`, que representan sus principales características. El comando `Object` proporciona acceso a la clase `Object`, que a su vez contiene todas las características comunes a todos los objetos y nos permiten definir objetos nuevos. Los objetos que se definen son inicialmente instancias de esta clase `Object`. Existen también los conceptos de constructor y destructor similares a los poseídos por otros lenguajes. La clase *Object* posee un conjunto de operaciones que le permiten manipular las estructuras de datos que poseen (modificar y obtener valores de variables, etc.). Estas operaciones permiten también crear atributos nuevos, lo que ocurre automáticamente si se intenta escribir sobre alguno que no exista previamente (operación `set`). también existe la operación `unset`, que permite borrar un conjunto de atributos sobre un objeto dado. *XOTcl* cuenta por tanto con capacidades de reflexión estructural.

Los métodos en *XOTcl* están construidos de forma similar a los procedimientos de *Tcl*. Los objetos pueden definir métodos propios (llamados *procs*), y si éstos son llamados por las clases se denominan *instprocs*. Se pueden definir métodos especificando nombre, argumentos y cuerpo (código) dinámicamente. A la hora de crear código para estos métodos (o un método cualquiera), se pueden usar los comandos `self` (el nombre del objeto que está en ejecución, `self class` (la clase que está actualmente ejecutando el *instproc*) y `self proc` (el nombre del método que está actualmente ejecutándose). Éstos son ejemplos de las capacidades reflexión estructural y también de introspección que el lenguaje ofrece. Todo método definido mediante `proc` puede eliminarse usando el mismo comando especificando una lista de argumentos y código vacíos:

```
Habitacion proc entrar {} {}
```

En lo relativo a las clases, hay varias formas diferentes de crearlas dentro de *XOTcl*. En cualquier caso las clases siempre derivarán de una metaclase. El comando `Class` visto anteriormente siempre proporciona acceso a la metaclase `Class`, que guarda todas las características comunes a todas las clases y permite derivar nuevas metaclases. El proceso de creación de clases es análogo al descrito para objetos. A través de la definición de una clase podemos crear instancias que pertenezcan a la misma.

Si se crea una variable sobre un objeto `Class`, esta variable será compartida por todas sus instancias (variable de clase). Los métodos que se definen en las clases y que se le proporcionan a las instancias son conocidos como *instprocs*. Es posible definir

métodos tanto en clases como en objetos por separado. Uno de estos métodos es precisamente el constructor. Un `instproc` puede ser eliminado de la forma ya vista. Por tanto, vemos como *XOTcl* ofrece soporte para reflexión estructural tanto a nivel de clase como a nivel de objeto, para todo tipo de miembros.

El modelo de herencia de *XOTcl* proporciona soporte para herencia simple y múltiple. Cada clase creada hereda de `Object` automáticamente, es decir, todas las clases pueden usar las características comunes a todos los objetos del lenguaje. Para especificar de qué clase hereda una dada se usa el siguiente comando:

```
# className -superclass classList
# Ejemplo
Class Habitacion
Class Cocina -superclass Habitacion
```

Haciendo esta operación, todas las instancias de `Cocina` serán capaces de acceder a las operaciones de `Habitacion`. *XOTcl* cuenta con un sistema para resolver colisiones entre nombres en caso de usar herencia múltiple. Las relaciones entre clases en *XOTcl* son dinámicas, es decir, en cualquier momento el grafo que determina las relaciones entre clases puede cambiarse completamente. Para ello se usa el comando `superclass` como se muestra en este ejemplo:

```
  Class SalidaTexto
  TextOutput instproc paint args {
    # pintar la salida ...
  }
  Class SalidaGrafica
  GraphicalOutput instproc paint args {
    # pintar la salida ...
  }

  # Inicialmente usamos salida por texto
  Class VirtualWorldOutput -superclass TextOutput
  VirtualWorldOutput instproc paint args {
    # hacemos las operaciones necesarias antes de pintar ...
    # pintamos
  }

  # El usuario decide cambiar la salida en texto por una gráfica
  VirtualWorldOutput superclass GraphicalOutput
```

Este tipo de cambios no sólo se permite para todas las instancias de una clase (cambiar la jerarquita de las clases), sino que un objeto concreto puede también cambiar dinámicamente de clase.

Como hemos visto, otro concepto incorporado por *XOTcl* son las metaclases. Éstas son un tipo especial de clases que se encargan de manejar la definición de clases. Toda clase es creada por una metaclase (normalmente la metaclase `Class`) y se pueden crear nuevas metaclases si el usuario lo desea. Las metaclases pueden usarse para contener partes abstractas comunes de las estructuras, para dar más capacidades a unas clases que a otras, para ciertas funcionalidades a nivel de clase (contar las instancias de una clase concreta) o bien para limitar funcionalidades, como por ejemplo impidiendo que puedan obtener información acerca de si mismas, como se ve en el siguiente ejemplo:

```
Class NoClassInfo -superclass Class

# Redefinimos el comando de introspeccion
NoClassInfo instproc info args {
    error "class info no disponible"
}
```

```
# Derivamos la clase Agente de la metaclase anterior, que no permite acceder a info
NoClassInfo Agent

# Ahora una llamada como:
# Agent info superclass
# lanzará un mensaje de error.
```

Además, el lenguaje cuenta con características de introspección, permitiéndole obtener el conjunto de variables locales, procedimientos, argumentos de procedimientos y el código de los mismos. La mayoría de los elementos son accesibles mediante el comando `info` que vimos en el ejemplo anterior, usando para ello la información contenida en objetos o metaclases:

- Paquetes y Extensiones: Permite saber que versiones hay disponibles de un determinado paquete, sus dependencias, *namespaces*, variables, comandos, procedimientos, etc.
- Variables: Se puede listas las variables que existen, si son *arrays* y que actividades han tenido lugar con esa variable (trazas).
- Comandos: Se puede obtener una lista de los comandos que están definidos en un momento dado.
- Procedimientos: Existe soporte para la introspección dentro de un procedimiento, que permite la realización de herramientas de depuración muy potentes. Permite obtener argumentos, cuerpo del método, …
- Pila: Un programa puede manipular la pila actual. De esta forma, un procedimiento por ejemplo puede mirar que procedimiento le ha llamado (hasta un nivel potencialmente infinito de llamantes)
- Otros elementos: Es posible acceder también a espacios de nombres y algunas características propias del sistema adicionales.

Tcl posee además varias características avanzadas adicionales que lo hacen un lenguaje muy potente [Abelson06]. Un ejemplo de estas características es la capacidad del lenguaje para que un programa pueda escribir otro programa y solicitar su evaluación, lo que se hace mediante la llamada a la función `eval`, lo que lo dota de capacidad de generar dinámicamente código.

A modo de resumen, podemos decir que las capacidades de flexibilidad de *XOTcl* son elevadas, al soportar reflexión estructural completa, tener un modelo de objetos que soporta herencia dinámica y permitir la generación dinámica de código.

5.3 LENGUAJES DINÁMICOS DE PROPÓSITO GENERAL

Bajo esta sección se contemplan todos aquellos lenguajes dinámicos que, por sus características concretas o el uso que se le dan, no encajan dentro de la categoría de *scripting* anterior. Se recogen pues aquí todos los lenguajes dinámicos de propósito general y no vinculados a ninguna aplicación/función concreta que se han considerado más importantes. Dentro de esta categoría encajarían lenguajes como *Smalltalk, Python,*

Dylan y *Self*, que se han descrito ya en el capítulo destinado a reflexión, por lo que en este capítulo nos centraremos en aspectos como su modelo de objetos y capacidades de reflexión, ampliando la información dada anteriormente sobre los mismos.

5.3.1 Ruby

Además de lo mencionado acerca de *Ruby* en el capítulo destinado a reflexión estructural, profundizaremos en la descripción de este lenguaje haciendo especial hincapié en su modelo de objetos y en sus capacidades de reflexión. El modelo de objetos empleado por este lenguaje está basado en el modelo de prototipos. Aunque existe la palabra reservada `class`, ésta funciona de manera distinta respecto al modelo de orientación a objetos basado en clases de lenguajes como *C++* [Tldp06]. *Ruby* agrupa el comportamiento compartido por una serie de objetos en una entidad que denomina clase, que contendrá por tanto todos los métodos comunes a los mismos (es decir, las clases funcionan como los *trait objects* mencionados en el capítulo anterior). Por ejemplo, podemos crear una clase `Perro` que contenga un comportamiento común a todos sus objetos, como el ladrido [Tldp06]:

```
class Perro
   def ladra
           print "guau, guau\n"
   end
end
```

Para usar este comportamiento, debemos declarar un objeto asociado a la clase recién definida, a través del cual podemos llamar al método `ladra`:

```
brian = Perro.new
```

Por tanto, sólo a través de los objetos podremos acceder al comportamiento compartido definido en su clase. Como en este lenguaje las clases sólo contendrán el comportamiento compartido por los objetos, son estos últimos los encargados de guardar su estado, a través de las denominadas variables de instancia. Una variable de instancia [Tldp06] tiene un nombre que comienza con `@`, y su ámbito está limitado al objeto al que referencia *self* (equivalente a *this* en *C++*, refiriéndose la propia instancia). Dos objetos diferentes, aun cuando pertenezcan a la misma clase, pueden tener valores diferentes en sus variables de instancia, tal y como ocurre en los lenguajes orientados a objetos "tradicionales". En *Ruby* las variables de instancia nunca son públicas y tienen el valor `nil` antes de que se inicialicen. Las variables de instancia en *Ruby* no necesitan declararse, proporcionándose de esta forma una flexibilidad mayor a los objetos, ya que cada una de ellas se añade dinámicamente al objeto la primera vez que se la referencia, como se muestra en el siguiente ejemplo:

```
class InstTest
   def set_foo(n)
           @foo = n
   end
   def set_bar(n)
           @bar = n
   end
end
nil
i = InstTest.new
```

```
#<InstTest:0x401c3e0c>
i.set_foo(2)
2
i
#<InstTest:0x401c3e0c  @foo=2>
i.set_bar(4)
4
i
#<InstTest:0x401c3e0c  @bar=4, @foo=2>
```

A tenor de todo lo visto, podemos concluir que *Ruby*, aun manteniendo una nomenclatura similar al modelo de orientación a objetos basado en clases tradicional, emplea conceptos del modelo de prototipos. De esta forma, las clases contendrán todo el comportamiento compartido de las instancias solamente (serán un *trait object*, como se vio anteriormente), mientras que el estado de cada instancia estará contenido en cada una de ellas individualmente. La modificación de la estructura de una instancia no hará caer al modelo en un estado inconsistente, puesto que la clase no contiene ningún dato acerca del estado y cualquier método añadido a una instancia será propiedad de la misma, sin alterar la estructura del resto. Un objeto puede así redefinir el comportamiento de un método de su clase, haciendo que ese objeto tenga un comportamiento específico, que sólo tendrá efecto para dicho objeto (método *singleton*), sin necesidad de crear una nueva clase para el mismo. Por tanto, a tenor de lo visto, podemos concluir que los objetos de una clase no tienen porque tener la misma estructura que la misma.

El modelo de herencia de *Ruby* está basado en delegación, aunque no permite la herencia dinámica (no se puede cambiar la clase de un objeto ni el árbol de herencia de las mismas). Tampoco se soporta la herencia múltiple y todo objeto posee como ancestro `Object`. La carencia de herencia múltiple se suple con la introducción del concepto de módulo y de *mixin* (incluido en el concepto de módulo), que describiremos a continuación.

Los módulos son un modo de agrupar clases, métodos y constantes y evitar de esta forma la colisión de identificadores. Pueden poseer código que se ejecuta cuando se cargan. El concepto de módulo está acompañado del concepto de *mixin*. Un *mixin* colecciona una funcionalidad que está relacionada entre si, y que es utilizada mediante herencia de implementación. Cuando una clase incluye un módulo, los métodos de instancia de este módulo se convierten en métodos de instancia de esa clase. El mismo módulo puede ser incluido en diferentes clases y puede aparecer en distintos puntos de la jerarquía de herencia. Por otra parte, si se modifica la definición de un método en un módulo, la definición se cambiara en cualquier clase que lo incluya. Los módulos pueden incluir otros módulos. A continuación se muestra un ejemplo muy simple de cómo se trabaja con ellos:

```
module ModuloSimple
  def hola
    "Hola"
  end
end
class ClaseSencilla
  include ModuloSimple
end

s = ClaseSencilla.new
s.hola          »       "Hola"

module ModuloSimple
  def hola
    "Que tal?"
  end
end

s.hola          »       "Que tal?"
```

En lo referente a sus capacidades reflectivas, *Ruby* posee además soporte para introspección [RubyCentral06b]. Usando este mecanismo, *Ruby* permite examinar dinámicamente todos los objetos de un programa, su jerarquía de clases, atributos y métodos (y la información asociada a ellos). Para ilustrar estas capacidades, mostraremos dos ejemplos de uso de la introspección en este lenguaje. El siguiente código muestra cómo se puede obtener una lista de métodos que pueden ser llamados sobre una instancia:

```
r = 1..10 # Creamos un objeto Range
list = r.methods
list.length    »       60
list[0..3]     »       ["size", "end", "length", "exclude_end?"]

#Se chequea si un objeto soporta un determinado metodo
r.respond_to?("frozen?")      »       true
r.respond_to?("hasKey")       »       false
"me".respond_to?("==")        »       true

#Obtenemos la clase de un objeto y su id unico. Usando este id podemos obtener la #relacion
con otras clases
num = 1
num.id  »       3
num.class       »       Fixnum
num.kind_of? Fixnum    »       true
num.kind_of? Numeric   »       true
num.instance_of? Fixnum        »       true
num.instance_of? Numeric       »       false
```

Podemos también navegar por la jerarquía de clases definida en un momento concreto en el programa, como muestra el siguiente código:

```
klass = Fixnum
begin
  print klass
  klass = klass.superclass
  print " < " if klass
end while klass
puts
p Fixnum.ancestors

#Este código produce:

» Fixnum < Integer < Numeric < Object
» [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
```

Para ilustrar las capacidades de reflexión estructural de *Ruby* vistas anteriormente, mostraremos dos ejemplos a continuación [Ruby06]. En este primer ejemplo usaremos el concepto de módulo ya visto para añadir dinámicamente los métodos que se definan dentro del mismo a un objeto existente. Para ello se va a usar *Object#extend*:

```
module Humor
  def riete
    "je, je!"
  end
end

a = "Gracioso"
a.extend Humor
a.riete         »       "je, je!"
```

Si se usa *extend* con una definición de clase, los métodos del módulo automáticamente pasan a formar parte de la misma:

```
module Humor
  def riete
    "je, je!"
  end
end

class Gracioso
  include Humor
  extend  Humor
end

Gracioso.riete        »      "je, je!"
a = Gracioso.new
a.riete       »      "je, je!"
```

Vemos por tanto cómo a partir de una clase es posible añadirle nuevos métodos mediante este mecanismo. También podemos eliminar métodos con primitivas como *remove_method*, o que una clase particular deje de poder recibir llamadas al mismo con *undef_method*. En el caso de atributos, podemos eliminarlos mediante *remove_instance_variable* y *remove_class_variable*, según queramos eliminarlos de instancias o clases.

El segundo de los ejemplos que se mostrarán describe como *Ruby* permite crear una clase asociada a un objeto concreto, mediante la cual podremos redefinir el comportamiento de los métodos de la clase a la que perteneciese dicho objeto. En el ejemplo asociamos a un objeto *string* una clase anónima que redefinirá uno de los métodos que ya posee y añadirá uno nuevo, viéndose claramente como el comportamiento nuevo sólo se aplica sobre el objeto concreto:

```
a = "hola"
b = a.dup

class <<a
  def to_s
    "El valor es '#{self}'"
  end
  def twoTimes
    self + self
  end
end

a.to_s »      "El valor es 'hola'"
a.twoTimes    »      "holahola"
b.to_s »      "hola"
```

Ruby permite además que el usuario pueda capturar ciertos eventos del sistema mediante el empleo de *hooks*. Estos *hooks* permiten interceptar eventos como la creación de un objeto, lo que hace que podamos redefinir que hacen ciertas primitivas, cambiando efectivamente la semántica de las mismas y simulando de esta forma algunas capacidades de reflexión computacional. El código asociado a estos *hooks* puede hacer también funciones relativas a la reflexión estructural: añadir métodos nuevos al objeto sobre el que se llama, alterar alguna información en el mismo, etc., entre otras funciones. A modo de ejemplo se muestra aquí una captura y redefinición de la primitiva *new*, que permite crear nuevas instancias de un objeto:

```
class Class
  alias_method :old_new,  :new
  def new(*args)
    result = old_new(*args)
```

```
    result.timestamp = Time.now
    return result
  end
end
```

Vemos cómo la definición de esta nueva primitiva `new` asocia a cada objeto un atributo que indica la hora exacta en la que fue creado. Para esto se necesita añadir un atributo `timestamp` a cada `Object`, lo que podemos hacer usando la reflexión estructural incorporada en *Ruby*:

```
class Object
  def timestamp
    return @timestamp
  end
  def timestamp=(aTime)
    @timestamp = aTime
  end
end

# Esto produce
class Test
end

obj1 = Test.new
sleep 2
obj2 = Test.new

obj1.timestamp        »       Sun Jun 09 00:09:45 CDT 2002
obj2.timestamp        »       Sun Jun 09 00:09:47 CDT 2002
```

Existen formas adicionales de capturar estos eventos que ocurren dentro de un programa dentro de su ejecución. Mediante el empleo de *callbacks* podemos capturar ciertos eventos de forma controlada. Los eventos a capturar son (donde :: indica un método de clase o módulo y # indica un método de instancia):

Evento	**Método *Callback***
Añadir un método de instancia	Module#method_added
Añadir un método singleton	Kernel::singleton_method_added
Crear una subclase para una clase	Class#inherited
Incorporar un módulo a una clase	Module#extend_object

En tiempo de ejecución estos métodos serán llamados por el sistema cuando ocurra el evento especificado. Por defecto estos métodos no ejecutan operación alguna, pero si se define código para los mismos automáticamente pasará a ser ejecutado. Además, si por ejemplo tuviésemos implementada alguna operación que se ejecuta cada vez que un método se llame, entonces ahora se podrá aplicar dicha operación con los métodos que se añadan nuevos a cada clase/objeto.

A modo de conclusión, cabe destacar cómo el lenguaje *Ruby* permite al programador contar con un elevado grado de flexibilidad, abarcando tanto introspección como reflexión estructural (limitada, al no tener herencia dinámica) y ciertos elementos de reflexión computacional (de forma simulada con los *hooks*), sobre un modelo computacional basado en prototipos. También soporta la generación dinámica de código que le permite ampliar y modificar las aplicaciones en tiempo de ejecución con código

que se genere según el estado de la aplicación, respondiendo dinámicamente a posibles cambios que pudiesen ocurrir.

Por tanto, este lenguaje, al igual que otros que también describiremos (como *Python*), mantiene la palabra reservada `class`, empleada por el modelo "tradicional" de clases, para lograr una mayor familiaridad con los programadores acostumbrados al mismo (aunque como hemos visto su significado es muy diferente).

5.3.2 CLOS

CLOS [DeMichiel87] es el acrónimo de *Common Lisp Object System*, un lenguaje de programación orientado a objetos que forma parte de *Common Lisp* [LispWorks06]. *CLOS* posee una serie de características que difieren de la mayoría de lenguajes orientados a objetos tradicionales. El modelo poseído por este lenguaje es muy similar al que también posee *Dylan*. *CLOS* introduce el concepto de clase como entidad que determina la estructura y comportamiento de un conjunto de objetos (instancias), por lo que está basado en un modelo de clases tradicional [DeMichiel87]. Todo objeto debe ser instancia de una clase. Las clases en *CLOS* pueden heredar la estructura y comportamiento de otras clases y son a su vez instancias de otras clases (metaclases).

La existencia de metaclases en este lenguaje permite controlar también la estructura y el comportamiento del sistema de clases. Las funciones genéricas que el lenguaje posee son también instancias de clases. Los tipos predefinidos en *Common Lisp* dependen de la metaclase `standard-type-class` y no pueden ser instanciados con `make-instance` ni incluidos como clase padre de ningún otro tipo. Todos los tipos definidos por el programador son instancias de la metaclase `standard-class`.

En *CLOS* las clases contienen *slots,* con un nombre y un valor asociado. La clase controla el nombre, el número, y la forma de acceder a estos *slots*. Una clase se define usando la macro `defclass` e indicando todos los datos necesarios para construirla, al igual que ocurre con los *slots* que la forman. Existen dos tipos de *slots*: los que son locales a una instancia particular y aquéllos que son compartidos por todas las instancias de una clase data, algo que se determina con una opción al crear el *slot* denominada `:allocation`.

Como norma general, los *slots* son heredados por las subclases, salvo que la subclase tenga un *slot* que lo oculte. `defclass` también permite la generación de métodos que permitan leer y escribir *slots* si así se desea. A modo de ejemplo, se muestra el siguiente código que crea una clase `x-y-posicion` que hereda de la clase `posicion`:

```
(defclass posicion () ())
(defclass x-y-posicion (posicion)
((x :initform 0)
 (y :initform 0))
(:accessor-prefix posicion-))
```

Las operaciones específicas de clases en *CLOS* se implementan a través de funciones genéricas. Una función genérica es una función cuyo comportamiento depende de las clases de los argumentos que se le pasan, siendo los métodos asociados con esta función genérica los que definen las operaciones más especificas según la clase de esos argumentos (especializan la función genérica), concepto que ya ha sido descrito en un capítulo anterior con el lenguaje *Dylan*. Los métodos se definen mediante `defmethod` y se heredan también al igual que los atributos.

CLOS incorpora capacidades para redefinir las clases. Cuando un `defclass` se evalúa, y el nombre asociado a la clase ya existe, dicha clase existente es redefinida y se actualiza según lo especificado por el nuevo `defclass`. Redefinir una clase modifica completamente el objeto `class` existente para que cumpla con la nueva definición, y estos cambios son propagados a sus instancias y a las instancias de sus subclases de la forma especificada por la implementación con la que se esté trabajando (de manera ansiosa o perezosa). Actualizar una instancia no cambia su identidad, es decir, no se crea una nueva instancia aunque se cambien todos sus *slots*. Las instancias también admiten la creación de métodos *singleton* tal y como se han visto en *Ruby* anteriormente.

Los programadores pueden definir métodos dentro de la función genérica `class-changed` para controlar el proceso de redefinición mencionado, ya que esta función será invocada automáticamente cuando esta situación ocurra. Esta funcionalidad es parte del *MOP* (*Meta Object Protocol*) que *CLOS* implementa, y que le permite controlar varios aspectos del sistema y del lenguaje a través de un conjunto de metaobjetos. Este *MOP* permitirá crear otras formas de manipular objetos, definir otros objetos y en definitiva cambiar el modelo de objetos de acuerdo con una serie de necesidades, dotándole de soporte para reflectividad computacional. El *MOP* define una serie de clases que controlan la implementación de todo el resto de clases en el que el sistema se basa, y que el programador puede modificar añadiéndoles más *slots*, por ejemplo. Algunas de estas clases son:

- ***standard-class***: La clase por defecto de las clases definidas por `defclass`.
- ***standard-method***: La clase por defecto de los métodos definidos por `defmethod`.
- ***standard-generic-function***: La clase por defecto de las funciones genéricas definidas por `defmethod` y `defgeneric-options`.

Una vez enunciadas someramente las características principales de su modelo de objetos, podemos hacer un resumen de las capacidades del lenguaje mediante los siguientes puntos:

- No posee soporte para encapsulación, es tarea del sistema de gestión de paquetes.
- Soporta la herencia múltiple, pudiéndose combinar los métodos de la forma que el programador desee. Existe también un mecanismo de resolución de conflictos ante el uso de este tipo de herencia.
- La estructura de los objetos puede ser modificada en tiempo de ejecución. Para ello, *CLOS* permite cambiar la definición de las clases así como cambiar la clase a la que pertenece una determinada instancia. Se soporta por tanto reflexión estructural a nivel de clases y herencia dinámica (un objeto cualquiera puede también modificar la clase de la que es instancia dinámicamente, usando el operador `change-class`), siguiendo el procedimiento de actualización descrito anteriormente.
- No es necesario declarar un tipo para los atributos o variables del programa, su sistema de tipos es dinámico.
- Los métodos son implementados a través de funciones genéricas, similares a las mencionadas en el lenguaje *Dylan*. Además, posee "multimétodos", es decir, todos los argumentos pueden ser usados para seleccionar un objeto al que aplicarle el método, en lugar de sólo el primero como ocurre en otros lenguajes.

En lo referente a métodos, como ya se ha mencionado podemos encontrar tanto

funciones genéricas como métodos, de manera muy similar a los vistos en *Dylan*. Un método es conceptualmente muy similar en este lenguaje a una función, diferenciándose en que un método sólo se ejecutará si los argumentos encajan con los tipos declarados en la lista de parámetros. Otras características relativas a los métodos son:

- Se puede definir cualquier número de métodos con el mismo nombre pero con diferente lista de parámetros, escogiendo el sistema el que más se ajuste a los argumentos pasados y usando todos ellos para determinarlo.
- Existen métodos que se pueden llamar antes, después o en vez de un método determinado, declarándolos explícitamente como métodos que cumplen este propósito, siendo un procedimiento similar al visto en el caso de la *AOP*.
- Todos los métodos de una función genérica (que agrupa métodos con el mismo nombre) deben tener el mismo número de argumentos obligatorios y opcionales.
- Hay métodos de acceso (estilo *get/set*) para los atributos, opcionales.

Por tanto, hemos visto cómo este lenguaje implementa un modelo de objetos basado en clases que soporta reflexión estructural a nivel de clases (cambiando de forma dinámica a las instancias de acuerdo con los cambios realizados en las clases: evolución de esquemas), y que le permite la implementación de un soporte (limitado) de reflexión estructural y computacional, usando un *MOP* para ésta última.

5.3.3 Dylan

Dylan es un lenguaje dinámico funcional, orientado a objetos, basado en clases y de propósito general que tiene un modelo de objetos que guarda una gran similitud con el lenguaje *CLOS* ya descrito. Las principales características de *Dylan* ya han sido expuestas en el capítulo dedicado a reflexión, por lo que no serán descritas de nuevo aquí. Las clases de este lenguaje son usadas para los siguientes fines:

- Son los tipos de datos y determinan las relaciones subtipo-supertipo entre los objetos.
- Permiten la abstracción de los atributos y métodos comunes de los objetos. Las subclases heredan los miembros asociados a las superclases.
- Permiten la especialización de los comportamientos que se aplicarán sobre los objetos, ya que una subclase puede definir nuevos comportamientos.

Los objetos guardan sus datos en *slots* mientras que el comportamiento asociado a los mismos reside en funciones genéricas y métodos, tal y como vimos en *CLOS*, funcionando estos elementos de la misma forma. Como en el lenguaje *CLOS*, las funciones genéricas son interfaces abstractos para operaciones concretas, de manera que se pueda cambiar la implementación de una operación sin cambiar el *interface* de la misma.

Dylan también incorpora el concepto de espacio de nombres a través de los módulos. Un módulo puede incluir o usar otros módulos, pero sólo las variables que se exporten explícitamente desde ellos son visibles. Los módulos permiten por tanto tener variables globales, públicas o privadas, y tener una interfaz para una colección de clases

y funciones genéricas. En *Dylan* todo es un objeto (incluso tipos simples, métodos y funciones genéricas). Por tanto, todos los elementos del lenguaje tendrán identidad propia, serán instancias de una clase y podrán manipularse de forma similar.

En lo relativo a herencia, cada clase podrá heredar de una o más clases. Toda clase derivará directa o indirectamente de `<object>`, que es el "padre" de todas las clases. Este lenguaje también posee la capacidad de crear métodos *Singleton* para instancias concretas [Apple98], que funcionan como se han descrito para otros lenguajes anteriormente.

La similitud del modelo de objetos de este lenguaje con el que tiene *CLOS*, hace que no nos extendamos en su descripción. Únicamente mencionaremos que *Dylan* limita las capacidades de flexibilidad poseídas por *Lisp* y *CLOS*, lenguajes en los que se inspiró, soportando principalmente características de introspección.

5.3.4 Python

Describiremos a continuación las características del modelo de objetos (basado en prototipos) y del soporte para reflexión que posee este lenguaje, complementando lo mencionado en el capítulo de reflexión estructural. *Python* es un lenguaje con un sistema de tipos dinámico (soporta el concepto de *duck typing* ya visto), por lo que son los valores, y no las variables en sí, los que tienen asociado un tipo. *Python* cuenta con un mecanismo de resolución dinámica de nombres, mediante el cual los nombres de métodos y variables son asociados al contenido al que realmente se refieren en tiempo de ejecución, en lugar de durante la compilación del programa. *Python* también soporta el concepto de *closure* mencionado anteriormente.

Sin embargo, lo más importante en este lenguaje es su capacidad para soportar reflexión estructural. Aunque *Python* usa efectivamente la palabra reservada `class`, las clases en *Python* no se comportan como las de un lenguaje orientado a objetos basado en clases como *C++*, sino que son *trait objects* tal y como vimos para *Ruby*. Tanto objetos como clases permiten que se le añadan o eliminen atributos o métodos a su definición, algo que no sería admisible si el modelo de objetos se comportase como el modelo de clases tradicional, al poder existir clases cuyas instancias tuviesen una estructura diferente de la suya. Por tanto, *Python* permite la existencia de atributos y métodos de clase y de instancia por separado, pudiendo pues añadir libremente a cada uno de estos elementos cualquier tipo de miembro.

Al conjunto de miembros definidos en el contexto de una clase, cuando ésta se crea o bien asignados posteriormente durante la ejecución, puede accederse a través de la propiedad `__dict__` (que también puede usarse sobre instancias). Podemos también definir por asignación un atributo dentro del contexto de una instancia individual, incluso con el mismo nombre que alguno de su clase, al que entonces ocultará:

```
class Pruebas:
        att1 = 1
p = Pruebas()
Pruebas.att1 => 1
p.att1 => 1
p.att1 = "instancia"
Pruebas.att1 => 1
p.att1 => "instancia"
```

Es posible acceder a un atributo de clase desde una instancia de la misma, a través de la propiedad de todo objeto *Python* que permite acceder a su clase

(__class__), tal y como se ve en el siguiente código:

```
Pruebas.att1 => 1
p.__class__.__dict__[att1] = "clase"
Pruebas.att1 => "clase"
p.att1 => "instancia"
```

En *Python*, los atributos definidos para una clase podrán leerse desde cualquier objeto de la misma, pero cuando se modifique su valor desde una de esas instancias, entonces pasará a tener una copia privada de los mismos (*copy on write*). Si posteriormente decidimos eliminar ese atributo de clase, la eliminación del mismo a través de una instancia (que tenga una copia privada) no afecta a la clase ni viceversa, es decir, los *trait objects* no condicionan el estado de los objetos y se mantiene el concepto de *trait object* dado anteriormente.

En *Python* todo tipo no simple es un objeto, incluidas las clases, cuya "clase" se denomina metaclase. El lenguaje soporta introspección de tipos y clases de manera que todo tipo podrá ser leído e inspeccionado, extrayendo sus miembros como un diccionario como hemos visto. No obstante, ya hemos visto que las capacidades reflectivas de *Python* pueden ir más allá, ya que se permite cambiar la estructura de clases y objetos en tiempo de ejecución (reflexión estructural). En concreto, *Python* ofrece, en tiempo de ejecución, un conjunto de funcionalidades que le permiten [Andersen98]:

- Modificar dinámicamente la clase de un objeto. Todo objeto posee un atributo denominado __class__ que hace referencia a su clase. La modificación del mismo implica la modificación del tipo del objeto.
- Acceder al árbol de herencia. Toda clase posee un atributo __bases__ que referencia una colección de sus clases base modificables dinámicamente. *Python* soporta herencia múltiple.
- Acceso a los atributos y métodos de un objeto. Hemos visto como tanto los objetos como las clases poseen un diccionario de sus miembros (__dict__) que puede ser consultado y modificado dinámicamente.
- Control de acceso a los atributos. El acceso a los atributos de una clase puede ser modificado con la definición de los métodos __getattr__ y __setattr__.
- Modificación de la estructura de clases y objetos: *Python* permite modificar la estructura de clases y objetos (añadir / modificar / eliminar atributos y métodos) simplemente mediante la asignación al objeto directamente de un valor y un nombre para un atributo, o sólo el nombre para un método, como se verá en un ejemplo posterior. Esto hace que *Python* implemente características de reflexión estructural de una forma muy sencilla.

El siguiente ejemplo muestra cómo se permite cambiar la jerarquía de clases de un determinado objeto [Jackson05]:

```
>>> class ClaseA:                    # Definimos ClaseA
      def setdata(self, value):
        self.data = value
      def display(self):
        print 'Clase A, data: %s' % self.data
>>> class ClaseB:                    # Definimos ClaseB
      def output(self):
        print 'Clase B, data: %s ' % self.data

>>> newobject = ClassA()             # Creamos una nueva instancia de ClaseA
>>> newobject.setdata(23)            # Damos a data el valor 23 para esa instancia
```

```
>>> newobject.display()              # Usamos el método de ClaseA
Class A data: 23

>>> newobject.__class__=ClassB       # Cambiamos la clase a ClaseB
>>> newobject.output()               # Usamos el metodo de ClaseB
Class B data: 23

>>> newobject.display()              # El metodo de ClaseA ya no existe
Traceback (most recent call last):   # Returns error
  File "", line 1, in ?
    newobject.display()
AttributeError: ClassB instance has no attribute 'display'

>>> newobject.__class__=ClassA       # Cambiamos otra vez la clase a ClaseA
>>> newobject.display()              # Usamos el metodo de ClaseA
Class A data: 23
```

Por tanto, *Python* permite la herencia dinámica y el cambio de tipo. Sobre la reflexión estructural del lenguaje se han construido módulos que aumentan las características reflectivas del entorno de programación [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91]. Para ilustrar también cómo se permite la modificación de la estructura de una clase, se muestra el siguiente ejemplo:

```
#Clase Point
class Point:
        # Constructor
        def __init__(self, x, y):
                self.x=x
                self.y=y

        #Método mover
        def move(self, relx, rely):
                self.x=self.x+relx
                self.y=self.y+rely

        #Método dibujar
        def draw(self):
                print "("+str(self.x)+","+str(self.y)+")"

#Programa
point=Point(1,2)
point.draw() # (1,2)

# Modificamos los atributos de un único objeto
point.z=3
print point.z # 3

# Modificamos los métodos de un único objeto, creando un método nuevo dinámicamente
def draw3D(self):
        print "("+str(self.x)+ "," + str(self.y) + "," + str(self.z) + ")"

point.draw3D=draw3D
point.draw3D() # (1,2,3)

# Modificamos los métodos de una clase, creando un método nuevo dinámicamente
def getX(self):
        return self.x

Point.getX=getX
print point.getX() # 1
```

En este código vemos cómo se puede cambiar fácilmente atributos y métodos tanto de una clase como de un objeto dinámicamente, en tiempo de ejecución, así como añadir nuevos miembros. Este ejemplo junto con el anterior forman una muestra de las posibilidades que ofrece la reflectividad estructural implementada en este lenguaje, permitiendo alterar la estructura de clases y objetos (añadiéndoles y borrándoles (con `del`) miembros), cambiar las relaciones existentes entre clases (herencia) y también

cómo el sistema no devuelve un error fatal en caso de que se acceda a un miembro no existente en un momento dado debido a este tipo de cambios, algo que permite reaccionar a este error de forma controlada y continuar la ejecución por una vía alternativa si así lo deseamos.

Por tanto, aunque este lenguaje parece que posee un modelo computacional basado en clases, una vez compilado su código fuente, en fase de ejecución, el modelo computacional empleado estará basado en el modelo de prototipos. En tiempo de ejecución las abstracciones de clases son substituidas por objetos que las representan. Vemos pues como, realmente, en *Python* las clases definidas son lo que en el modelo de prototipos se denomina *trait objects*, que contendrían todo lo que es compartido por sus instancias, permitiendo así a los objetos tener todos aquellos miembros que deseen de forma privada. Al igual que *Ruby*, el lenguaje posee la palabra reservada `class` como sinónimo de *trait object* y no como una clase en el modelo orientado a objetos basado en clases tradicional.

5.3.5 Self

Self es un lenguaje orientado a objetos cuyo modelo computacional está basado en prototipos y, como tal, carece completamente del concepto de clase. Se complementará en este apartado la descripción del lenguaje dada en un capítulo anterior, describiendo brevemente el modelo empleado por el lenguaje y sus peculiaridades, así como sus capacidades de reflexión estructural.

Los objetos en *Self* son una colección de "*slots*" [Agesen00]. Los *slots* pueden ser métodos de acceso que devuelven valores, o también un medio para modificar el valor asociado. Por ejemplo, un *slot* llamado "nombre" devolvería su valor así:

```
miPersona nombre
```

Si quisiésemos modificar ese valor deberíamos hacer:

```
miPersona nombre:'Brian'
```

Los métodos son objetos que contienen código, y pueden introducirse en *slots* de la misma forma que los atributos. De hecho, no hay forma de distinguir en este lenguaje entre atributos y métodos, ya que todo es tratado uniformemente. Todo objeto en este lenguaje puede ser una entidad aislada, al no existir ni clases ni metaclases relacionadas con el mismo para realizar cualquier tarea. Por tanto, las modificaciones a un objeto dado en principio no afectarían a ningún otro. Un objeto podrá sólo responder a los mensajes que estén definidos dentro de su ámbito, aunque se pueden identificar objetos "padre" en los que el objeto puede delegar la ejecución de cualquier mensaje al que no pueda responder por no tener definido, objetos que pueden cambiarse en tiempo de ejecución (posee pues herencia por delegación y dinámica).

De esta forma, un prototipo en este lenguaje puede definir la totalidad de la estructura y comportamiento de los objetos que se crearán a partir de él, creando así objetos con sus miembros ya definidos en su interior, o pueden existir opcionalmente *trait objects*, que contengan un determinado comportamiento común a múltiples objetos y pueda ser compartido por todos los objetos vinculados al mismo, sin necesidad pues de duplicar estos elementos compartidos en todas las instancias, en función de las normas establecidas por el propio lenguaje.

Dado que esta estructura es igualmente válida tanto para atributos como para métodos (al no existir una distinción sintáctica entre ellos), cada objeto será pues una entidad capaz de contener todos los miembros que requiera de forma individual sin vulnerar su modelo computacional, dado que el concepto de clase, tal y como se define tradicionalmente, no existe en este lenguaje. De esta forma se pueden implementar operaciones reflectivas que permitan modificar los miembros de cualquier objeto con total libertad, al no haber dependencias de otras entidades como en el caso de las clases, algo que ya se menciono anteriormente. Por último, cabe decir que el modelo de *Self* es una evolución del presentado por *Smalltalk*, aunque eliminando el concepto de clase que este lenguaje posee, implementado de forma similar a otros lenguajes que se han visto en este mismo capítulo.

Self 4.0 cuenta con su propia interfaz gráfica para el acceso y manipulación de los miembros de los objetos del sistema en cada momento, pero esto mismo puede hacerse a través de una serie de primitivas que el lenguaje define para ello. *Self* posee un tipo de objeto especial denominado *mirror* que es el medio por el que el lenguaje hace uso de sus capacidades de reflexión [Wolczko06]. Estos objetos permiten encontrar *slots* por su nombre, obtener el código de los métodos y otras operaciones. Un *mirror* se puede obtener a partir de cualquier objeto usando la primitiva `_Mirror`, denominándose el objeto al que designa como su reflejado. A continuación se muestran un conjunto de estas primitivas y un ejemplo de uso, que ilustre la forma de trabajo de este lenguaje, y que son un conjunto de posibilidades del objeto `abstractMirror` que el lenguaje posee:

- ***reflectee***: Obtiene el objeto que se está inspeccionando.
- ***size***: Número de *slots* del objeto reflejado.
- ***names***: Un vector de nombres de los *slots*.
- ***at: slotName PutContents: mirror****:* Cambia el contenido de *slotName* para que sea el objeto reflejado de `mirror`.
- ***parentsDo: aBlock***: Itera sobre los *slots* padre.
- ***addSlot: slotMirror***: Añade un *slot*.
- ***addSlots: mirror***: Añade todos los *slots* del objeto reflejado por *mirror*.
- ***addSlotsIfAbsent: mirror***: Idéntico al anterior, pero sólo si los *slots* no existen ya.
- ***define: mirror***: Hace que el objeto reflejado del receptor sea idéntico al objeto reflejado del argumento.

Un ejemplo de uso es el siguiente:

```
_AddSlots: (| d = (('a' @ 2) & ('b' @ 3)) asDictionary.
              o = (| a = 2. b = 3 |)
            |)
d at: 'b'                          "imprime 3"
d at: 'b' Put: 4
d at: 'b'                          "imprime 4"
o _Mirror at: 'b'                  "imprime b = 3"
o _Mirror at: 'b' Put: 4
o _Mirror at: 'b'                  "imprime b = 4"
(o _Mirror at: 'b') name: 'c'      "cambia el nombre del slot b a c"
o c                                "imprime 4"
```

A modo de curiosidad, decir por último que un *interface* de usuario desarrollado para *Self* originalmente, llamado *Morphic* [Squeak05], puede aplicarse ahora a ciertas

implementaciones de *Smalltalk*, como *Squeak* [Squeak05], y permiten a este lenguaje funcionar como si su modelo de objetos fuese más similar al de prototipos, como una alternativa a la interfaz de usuario "estándar" poseído por esta distribución.

5.3.6 Groovy

Groovy es un lenguaje de programación orientado a objetos para la plataforma *Java* [Glover04] [Volkmann03]. Se puede considerar un lenguaje de *scripting* para esta plataforma, aunque por sus características es capaz de crear aplicaciones finales concretas, por lo que no es en sí un lenguaje de *scripting* según lo descrito en esta tesis. Posee características similares a otros lenguajes como *Python*, *Ruby*, *Perl* y *Smalltalk,* que veremos en esta sección. Otro nombre por el que es conocido es *JSR 241*.

El compilador de *Groovy* genera *bytecode* estándar de *JVM* y por tanto este lenguaje se compila (dinámicamente) a esta plataforma directamente, permitiéndole usar las librerías existentes en *Java* (y el propio código *Java*) o que un programa *Java* pueda usarlo en cualquier parte de la aplicación en construcción. *Groovy* es un proyecto en desarrollo y actualmente está en fase de estandarización (*Java Community Process*, *JSR 241*). *Groovy* tiene las siguientes características principales:

- **Compilación dinámica**: Los programas en *Groovy* pueden ser compilador como un archivo *.class* de *Java* estándar o bien ser interpretados en tiempo de ejecución, lo que es de utilidad para la creación de prototipos rápidos o tareas de *testing*.
- Permite **tipos dinámicos**.
- Los programas en *Groovy* pueden ser usados dentro de aplicaciones *Java* "normales" sin dificultad.
- *Groovy* permite que sus programas prescindan del método `main` para ser ejecutados junto a una aplicación a la que se asocian.

El hecho de que *Groovy* posea tipos dinámicos permite que no se declaren los tipos de las variables y que el tipo sea inferido en tiempo de ejecución (concepto de *duck typing* ya mencionado anteriormente). Esta característica permite al lenguaje tener más flexibilidad que *Java* y la existencia de métodos que trabajen con atributos concretos sin estar limitados por el tipo del objeto que posee el atributo, tal y como se ve en el siguiente ejemplo, donde cualquier objeto que defina el atributo `nombre` podrá ser usado en la función:

```
class Disco{
  duracion
  nombre
}

class Libro{
  nombre
  autor
}

def identificarMaterial(objeto){
  println "Se está procesando un objeto cuya identificación es: " + objeto.nombre
}
```

Otra característica de *Groovy* es que todo elemento del lenguaje es un objeto, incluidas las funciones, como se ve en el siguiente ejemplo:

```
disco = new Disco(duracion:120, nombre:"Made in Heaven")
libro = new Libro(nombre:"El capitan Alatriste", autor:"A. Perez Reverte")

identificarMaterial(disco) //Imprime 'Made in Heaven'
identificarMaterial(libro) //Imprime 'El capitan Alatriste'

funcion = identificarMaterial

funcion(disco) //Imprime 'Made in Heaven'
```

A nivel de *bytecode*, las clases de *Groovy* son clases *Java* estándar. No obstante, a diferencia de *Java*, *Groovy* hace que por defecto todo lo que se defina en una clase sea público (salvo que se defina un modificador de acceso concreto). También permite la creación de funciones de primer orden que existen fuera de la definición de una clase, la manipulación de expresiones regulares de manera más potente, los operadores sobrecargados y los parámetros por omisión. Otra de las novedades destacables que *Groovy* incorpora respecto a *Java* es el concepto de *closure* ya visto para otros lenguajes dinámicos descritos en este capítulo. Éstos se han incorporado de una forma muy integrada con la sintaxis del lenguaje, como se ve en el siguiente ejemplo sencillo:

```
class Alarma{
  accion

  darAlarma(){
    accion.call()
  }
}

sonido = { println "Beep"} //El codigo podria ser mucho más complejo.
visual = { println "Encendiendo bombillas" } //El codigo podria ser mucho más complejo.

unaAlarma = new Alarma(accion:sonido)
unaAlarma.darAlarma()  // Imprime 'Beep'

otraAlarma = new Alarma(accion:visual)
otraAlarma.darAlarma() // Imprime 'Encendiendo Bombillas'
```

La presencia de *closures* suple la carencia (al menos hasta el momento) de *Groovy* de *nested classes* e *inner classes*.

Una vez descritas las principales características de *Groovy* y sus aportaciones más significativas respecto al lenguaje *Java*, hablaremos de su modelo de objetos y de sus capacidades de reflexión. Dado que *Groovy* es un lenguaje creado para la *JVM*, el modelo de objetos esta basado en clases de forma similar al poseído por *Java*, pero cuenta con una serie de diferencias que enumeraremos a continuación:

- **Las clases pueden ser modificadas**: *Groovy* permite que las clases puedan modificarse de dos formas. Una de ellas es usar el concepto de metaclase. Existe una clase *MetaClass* [Groovy06b] que contiene los métodos y atributos que la clase asociada ya tiene y permite acceder a ellos. Esta clase además permite añadir en tiempo de ejecución más métodos, haciendo que efectivamente se puedan añadir nuevos comportamientos a clases existentes. No obstante, para usar esta clase apropiadamente durante un programa es necesario inicializarla, y una vez que el proceso de inicialización ocurre ya no permite añadir más métodos. Otra de las formas que existen de modificar clases es el redefinir dinámicamente el código de un método, aunque *Groovy* lo permite hacer de forma limitada como se muestra en este ejemplo [Sundararajan06], donde el cambio del código sólo

funciona dentro del ámbito de *use*:

```
class Persona {
  def saluda() {
    decirHola()
  }
}

p = new Persona()

class Saludador {
  static def decirHola(obj) {
    println "Hola mundo!"
  }
}

use(Saludador) {
  p.saluda()
}
```

- **Extender instancias**: *Groovy* también permite extender instancias existentes de objetos (de forma limitada). Estas capacidades se pueden lograr mediante la manipulación de la metaclase asociada a la clase del objeto (definiendo un nuevo objeto que contenga funcionalidad adicional para el método llamado) o bien reasignando el código de un método si la clase hereda de *Expando*, a través de bloques de código y propiedades. Los siguientes dos ejemplos muestran cómo se hacen estas operaciones y también las limitaciones existentes al respecto [Sundararajan06]:

```
class Persona {
  def saluda() {
    decirHola()
  }
}

p = new Persona()
def pmc = ProxyMetaClass.getInstance(Persona.class);

class MiInterceptor implements Interceptor {
   def invoke

   Object beforeInvoke(Object object,String nombre, Object[] args) {
     if (nombre == "decirHola") {
       println "Hola Mundo!"
       invoke = false
     } else {
       invoke = true
     }
   }

   boolean doInvoke() {
     return invoke
   }

   Object afterInvoke(Object obj, String nombre, Object[] args, Object res) {
     return null
   }
}

pmc.interceptor = new MiInterceptor()
p.metaClass = pmc;
p.saluda()
```

```
class Persona extends Expando {
}

p = new Persona()
// añadimos una propiedad que es un bloque de código
```

```
p.saluda = { println "p dice hola"}
// la llamamos como un método
p.saluda()

p2 = new Persona()
p2.saluda = { println "p2 dice hola" }
p2.saluda()
```

- ***Mixins***: Existe una propuesta que describe la incorporación en *Groovy* de *mixins* en tiempo de compilación, ejecución, para crear métodos *singleton* y *mixin* con parámetros [McCallister04].
- **Reflexión**: *Groovy* conserva todas las capacidades de reflexión de *Java*, aunque con una diferencia sintáctica: donde en *Java* era necesario `objeto.getClass()` ahora es necesario `objeto.class`.
- ***MOP***: *Groovy* también incorpora el concepto de *MOP* ya visto [Glover05] [McClean06]. El concepto de *MOP* en *Groovy* se basa en que todos los objetos del lenguaje implementan implícitamente `groovy.lang.GroovyObject` [Groovy06], que define los métodos `invokeMethod()` y `set/getProperty()`. En tiempo de ejecución, si se pasa un mensaje a un objeto que no ha sido definido como una propiedad ni en su clase ni en su jerarquía, entonces se hará una llamada a estos métodos. Aunque este mecanismo es útil para tratar errores, puede ser usado para responder de la manera adecuada a cualquier mensaje que se le pase a un objeto, delegando la responsabilidad de ejecutar dicho mensaje al *MOP* en sí. El siguiente código ilustra el proceso descrito:

```
class MOPHandler {
  def invokeMethod(String metodo, Object params) {
    println "A MOPHandler le han pedido invocar: ${metodo}"
    if(params != null){
        params.each{ println "\tcon el parámetro ${it}" }
    }
  }

  def getProperty(String propiedad){
     println "A MOPHandler le han pedido la propiedad: ${propiedad}"
  }
}
def hndler = new MOPHandler()
hndler.holaMundo()
hndler.crearUsuario("Matusalen", 999, new Date())
hndler.nombre

// Esto imprime:
A MOPHandler le han pedido invocar holaMundo
A MOPHandler le han pedido invocar crearUsuario
        Con el parámetro Matusalen
        Con el parámetro 28
        Con el parámetro Fri Dec 08 14:26:30 EDT 2006
A MOPHandler le han pedido la propiedad nombre.
```

A modo de conclusión podemos afirmar que el modelo de objetos y de reflexión de *Groovy* tiene puntos comunes con el de *Ruby*, ofreciendo más flexibilidad que *Java*. De esta forma, aunque se use un modelo basado en clases tradicional, hemos visto cómo se permite la ampliación dinámica de clases (de forma muy limitada y sólo para métodos) y algunos mecanismos que permiten extender instancias (usando un concepto muy similar a los *singleton methods* de *Ruby*), lo que en general le dota de unas capacidades de reflexión estructural bastante limitadas. Por otra parte, también ha quedado patente que la presencia de un *MOP* dota al lenguaje de algunas capacidades de reflexión computacional, aunque sólo para las primitivas que el lenguaje permite redefinir (como `invokeMethod`, `getProperty` y `setProperty`).

5.4 CONCLUSIONES GENERALES

A lo largo de este capítulo hemos hecho una descripción de los representantes que se han considerado más significativos de la familia de lenguajes dinámicos, haciendo hincapié especial en su modelo computacional de objetos y sus características dinámicas y reflectivas, complementando lo descrito al respecto en el capítulo de reflexión. El elevado grado de flexibilidad que todos los lenguajes vistos poseen les permite responder a requisitos cambiantes o nuevas reglas de negocio que surjan en el desarrollo de una aplicación más fácilmente.

También es posible destacar ciertos aspectos comunes a los lenguajes analizados además de sus ventajas ya mencionadas. Por ejemplo, ninguno de los sistemas visto proporciona una implementación completa de la reflexión computacional, principalmente debido al coste que ello supondría en tiempo de ejecución y a que las ventajas que aportaría no justificarían la pérdida de rendimiento. También ha quedado patente en este sentido que todos los lenguajes descritos soportan reflexión estructural (aunque algunos poseen algunas limitaciones), que les permiten modificar dinámicamente los miembros de clases e instancias y alterar las relaciones existentes entre objetos y clases. Todo ello les confiere una gran flexibilidad, que es suficiente para las necesidades de flexibilidad del *software* actuales (sistemas como *Zope* o *Ruby on Rails* se apoyan en esta capacidad para aumentar la productividad de los programadores que trabajan con estos sistemas y automatizar tareas). Cabe destacar pues cómo, a pesar de que la reflexión estructural no es el nivel de reflexión que ofrece más flexibilidad, los lenguajes dinámicos vistos la implementan al considerar que es el mejor compromiso existente entre rendimiento y flexibilidad, por lo que este nivel de reflexión será el que tratemos de buscar en esta tesis.

En lo relativo a su modelo computacional, podemos concluir que, aunque los modelos computacionales de los lenguajes dinámicos se basan en el de prototipos, muchos de ellos (*Smalltalk*, *Python*, *Ruby*, *Dylan*) poseen la palabra reservada `class`, probablemente en un intento de que programadores acostumbrados a lenguajes basados en clases logren una mayor familiaridad con el mismo. No obstante, no debe confundirse el concepto de clase empleado por estos lenguajes dinámicos con el de lenguajes "estáticos" como *C++* o *Java*, ya que estas "clases" no exponen el comportamiento y la estructura de los objetos como hacen los estáticos, sino que solamente modelan comportamientos (compartidos por todos los objetos relacionados con la clase), y son los objetos los que guardan su propia estructura, pudiendo incluso contener además métodos con un comportamiento más específico privados a los mismos, tal y como hemos descrito en los lenguajes que usaban esta aproximación.

Por último, se deben mencionar los principales inconvenientes existentes en este tipo de lenguajes:

- **El rendimiento**: Como ya se ha mencionado, para poder lograr todas las características vistas es necesario que estos lenguajes sean implementados basándose en la interpretación de su código, lo que afecta negativamente este aspecto. Existen múltiples estudios en desarrollo para intentar mejorar la eficiencia de los lenguajes dinámicos (ámbito en el que se identifica esta tesis), siguiendo diferentes aproximaciones, como podrían ser las implementaciones de *Python Jython* [Jython06] e *IronPython* [Iron*Python*06], que presentan también problemas de rendimiento con primitivas reflectivas debido a la capa adicional de código introducida para emular sus capacidades reflectivas, o *Psyco* [Psyco06] y *PyPy* [PyPy06], proyectos en desarrollo para acelerar la ejecución de *Python* manipulando la forma en la que éste es compilado. Otras aproximaciones, como *Rite* [Davis06], están intentando usar una máquina virtual con el objetivo de lograr

un aumento de rendimiento del lenguaje *Ruby2*. La idea subyacente en la creación de esta última plataforma, aún en desarrollo, está más en consonancia con lo que se va a tratar de validar en esta tesis, donde usaremos también una máquina virtual (en este caso preexistente), altamente optimizada, de cara a demostrar que su modificación para soportar primitivas de reflexión de forma nativa permite aumentar el rendimiento de los lenguajes dinámicos que se implementen y ejecuten sobre la misma, generando código intermedio de dicha máquina.

- **Detección temprana de errores**: El sistema de tipos dinámico que poseen estos lenguajes hace que los errores de tipos se produzcan en tiempo de ejecución, dada la imposibilidad de detectarlos en la fase de compilación debido a la carencia de tipos de los datos que permitan hacer este tipo de comprobaciones. No obstante, ya se ha mencionado que una forma de disminuir el número de posibles errores de este tipo que se puedan producir es emplear programas de test automatizados, que ayuden a su detección prematura, como *PyUnit* [PyUnit04], *Junit* [Junit06] o *Nunit* [Nunit05]. Otra posibilidad es servirnos de analizadores estáticos de programas, como *PyCheker* [Laird02], que minimicen la aparición de posibles errores de esta clase.

6 MÁQUINAS ABSTRACTAS Y VIRTUALES

Siendo uno de los objetivos de esta tesis demostrar que la modificación de una máquina virtual para dar soporte nativo a primitivas dinámicas puede proporcionar una mejora en la eficiencia de ejecución de lenguajes dinámicos, debemos hacer una descripción de un conjunto representativo de máquinas virtuales actuales y las implementaciones concretas existentes de las mismas, distinguiendo tipos, características y contemplando todos aquellos aspectos de interés en este aspecto. La clasificación que se describe en este capítulo está basada en la ofrecida en [Ortin01], ampliándola donde sea necesario para reflejar nuevos avances y consideraciones.

6.1 PROCESADORES COMPUTACIONALES

Todo programa está formado por un conjunto ordenado de instrucciones que se dan al ordenador, indicándole las operaciones o tareas que se desea que realice [Cueva94]. Estos programas son ejecutados, animados o interpretados por un procesador computacional. Un procesador computacional por tanto ejecutará las instrucciones propias de un programa que accederán a los datos pertenecientes a dicho programa examinándolos o modificándolos. La implementación de un procesador computacional puede ser física (*hardware*) o lógica (*software*) mediante el desarrollo de otro programa que realizará las funciones descritas.

6.1.1 Procesadores Implementados Físicamente

Un procesador físico es un intérprete de programas que ha sido desarrollado de forma física (normalmente mediante un circuito integrado). Los procesadores más extendidos son los digitales síncronos. Están formados por una unidad de control, una memoria y una unidad aritmético-lógica, todas ellas interconectadas [Mandado73].

Un computador es un procesador digital síncrono, cuya unidad de control es un sistema secuencial síncrono que recibe desde el exterior (el programador) una secuencia de instrucciones que le indican las microoperaciones que debe realizar [Mandado73]. La secuencia de ejecución de estas operaciones es definida en la memoria mediante un programa, siendo la semántica de cada instrucción y el número global de instrucciones distintas posibles para cada procesador invariable. La principal ventaja de este tipo de procesadores frente a los lógicos es su mucha mayor velocidad, precisamente por haber sido desarrollados físicamente. No obstante, por este mismo motivo sufren el inconveniente de su falta de flexibilidad, ya que es mucho más complejo modificar una implementación basada en *hardware* para adaptarla a determinados requisitos.

6.1.2 Procesadores Implementados Lógicamente

Un procesador *software* es un programa que emula el funcionamiento de un determinado procesador, pudiendo interpretar programas destinados al mismo [Cueva98]. La modificación de un programa que emule a un procesador es mucho más sencilla que la modificación física de un procesador *hardware*, lo que hace que los emuladores *software* se utilicen, entre otras cosas, como herramientas de simulación encaminadas a la implementación física del procesador que emulan.

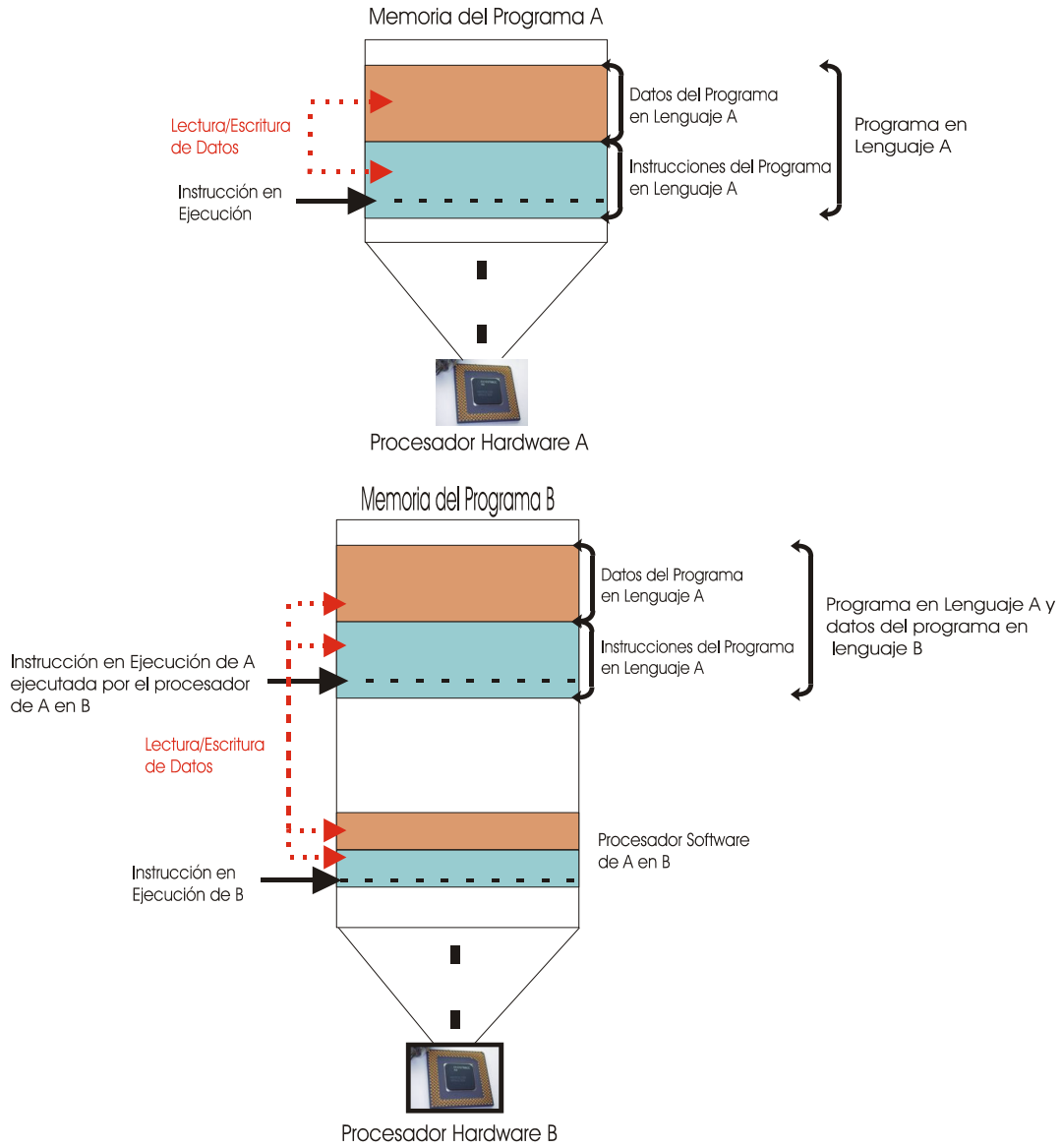

Figura 6.1: Ejecución de un procesador lógico frente a un procesador físico.

La principal desventaja de este tipo de procesadores frente a los procesadores *hardware* o físicos es la velocidad de ejecución. Dado que los procesadores lógicos establecen un nivel más de computación (son ejecutados o interpretados por otro procesador), requerirán por tanto un mayor número de computaciones para interpretar un programa que su versión *hardware*. Podemos resumir entonces diciendo que mientras

la implementación física de un procesador computacional proporciona el máximo nivel de eficiencia posible, dicho nivel de eficiencia se obtiene sacrificando la flexibilidad de dicho procesador, flexibilidad que sólo se puede obtener implementando dicho procesador mediante un programa *software*, que introducirá un nuevo nivel computacional y una sobrecarga mayor derivada del mismo.

Esta sobrecarga computacional se aprecia gráficamente en la Figura 6.1. En la parte superior de la figura se muestra cómo el procesador físico **A** va interpretando las distintas instrucciones máquina. La interpretación de las instrucciones implica la lectura y/o escritura de los datos. En el caso de interpretar a nivel *software* el programa, el procesador es a su vez un programa en otro procesador físico (procesador **B** en la Figura 6.1), existiendo pues una nueva capa de computación frente al ejemplo anterior. Esto hace que se requieran más computaciones[10] o cálculos que en el primer caso. En el segundo caso, el procesador **A** es más flexible que en el primero. La modificación, eliminación o adición de una instrucción de dicho procesador, se consigue con la modificación del programa emulador. Sin embargo, el mismo proceso en el primer ejemplo, requiere la modificación del procesador a nivel físico.

6.2 PROCESADORES LÓGICOS Y MÁQUINAS ABSTRACTAS

Una máquina abstracta es un diseño de un procesador computacional, pero que no tiene por que ser desarrollado de forma física [Howe99]. Apoyándose en dicho diseño del procesador computacional, se especifica formalmente la semántica del juego de instrucciones de la máquina, en función de la modificación del estado de la máquina abstracta. Un procesador computacional desarrollado físicamente es también una máquina abstracta, pero con una determinada implementación [Álvarez99]. Existen multitud de ejemplos de emuladores de microprocesadores físicos desarrollados como programas sobre otro procesador [SunPicoJ06]. Sin embargo, el nombre de máquina abstracta es empleado mayoritariamente para aquellos procesadores que no tienen una implementación física.

Un intérprete es un programa que ejecuta las instrucciones de un lenguaje que encuentra en un archivo fuente [Cueva98]. Su objetivo principal es animar la secuencia de operaciones que un programador ha especificado, en función de la descripción semántica de las instrucciones del lenguaje que se utilice. En unión con lo dicho anteriormente, un procesador computacional (implementado física o lógicamente) es también un intérprete del lenguaje que procese.

Una máquina virtual se define como un intérprete de una máquina abstracta[11] [Howe99]. Una máquina virtual es por tanto un intérprete definido sobre una máquina abstracta. De esta forma, la máquina abstracta es utilizada en la descripción semántica de las instrucciones de dicho intérprete.

En esta tesis sólo haremos mención a aspectos de las máquinas abstractas relacionados con su uso práctico y no mencionaremos otros aspectos más formales.

10 *Mayor número de computaciones no implica siempre mayor tiempo de ejecución. Esta propiedad se cumpliría si la velocidad de procesamiento del computador B fuese igual a la del computador A.*

11 Aunque el concepto de máquina abstracta y máquina virtual no son exactamente idénticos, son comúnmente intercambiados.

6.3 Uso del Concepto de Máquina Abstracta

En este punto se mencionarán distintas aplicaciones prácticas que se le han dado al concepto de máquina abstracta, especificando sus funcionalidades y una descripción colectiva de lo conseguido con cada utilización. En la sección "Panorámica de Utilización de Máquinas Abstractas" analizaremos cada caso particular, destacando sus aportaciones y carencias en función de los requisitos establecidos para esta tesis.

6.3.1 Procesadores de Lenguajes

El concepto de máquina abstracta se ha utilizado en la implementación de compiladores, con el objetivo de simplificar su diseño [Cueva94]. Para ello, el proceso de compilación tomaría un lenguaje de alto nivel generando a partir de él un código intermedio, siendo este código propio de una máquina abstracta. Esta máquina se diseñará lo más general posible, de forma que sea posible traducir de ésta a cualquier máquina real existente de manera sencilla. Para generar el código binario de una máquina real, tan sólo hay que traducir el código de la máquina abstracta a la máquina física elegida, independientemente del lenguaje de alto nivel que haya sido compilado previamente, logrando así un proceso común de traducción para todos los lenguajes que puedan traducirse previamente al código propio de la máquina abstracta.

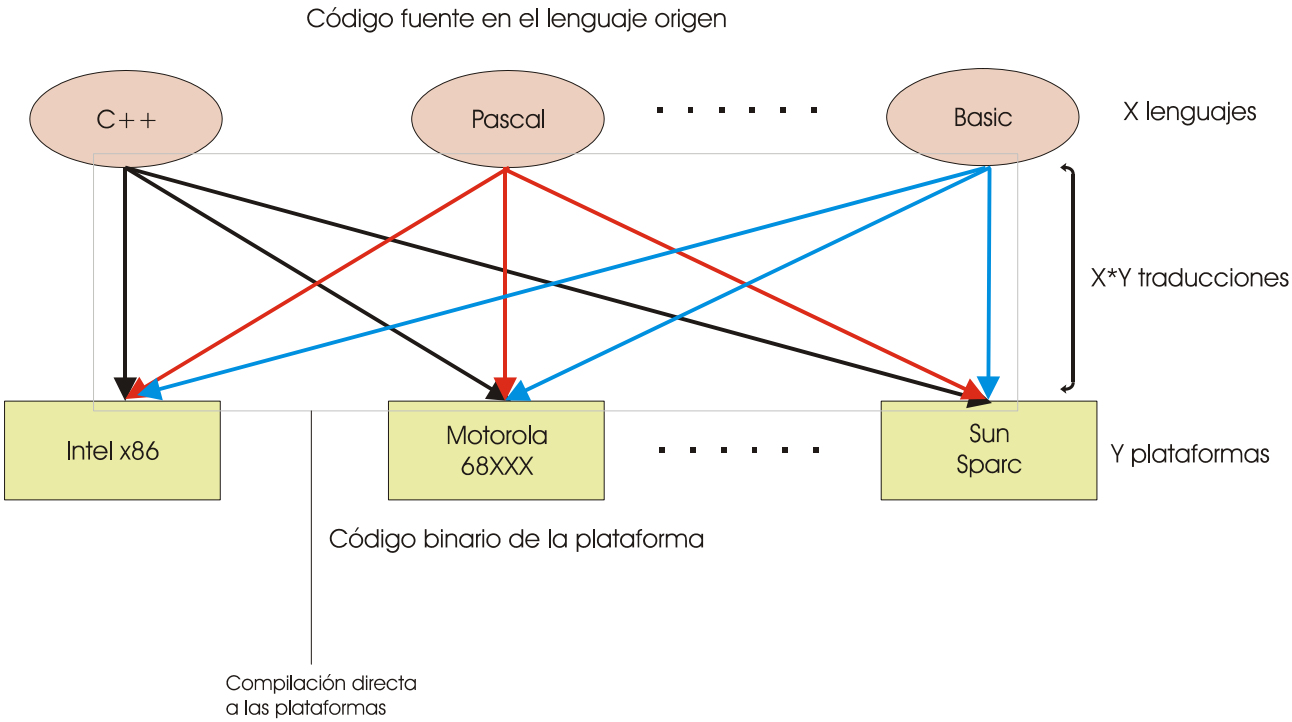

Figura 6.2: Compilación directa de N lenguajes a M plataformas.

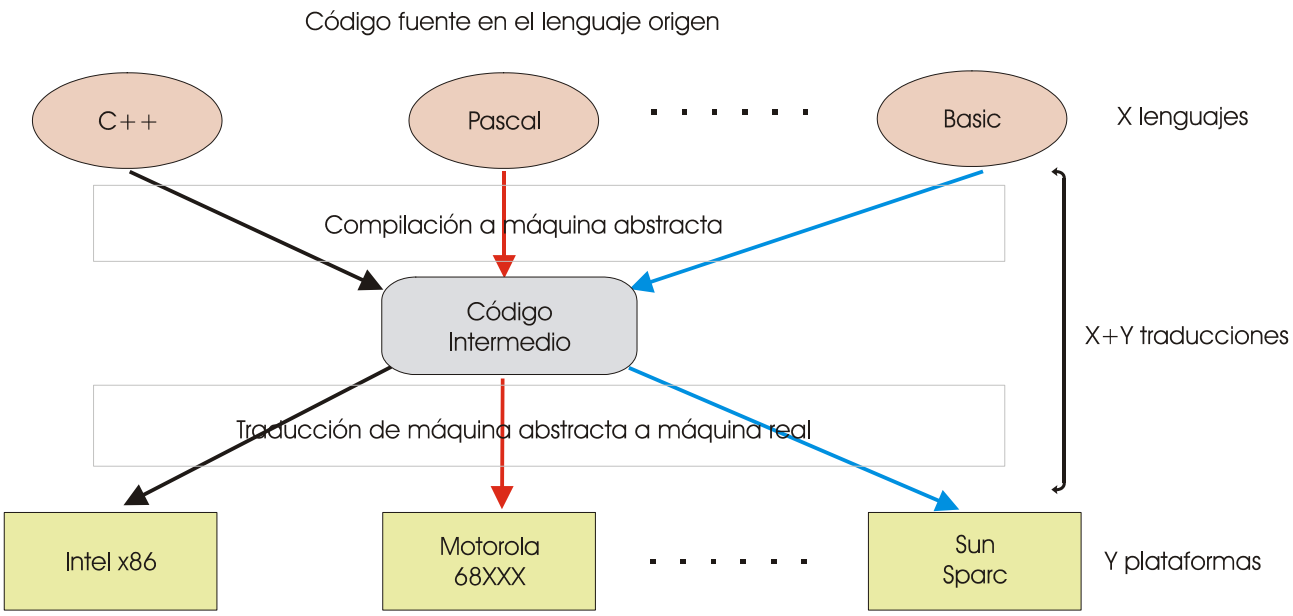

Figura 6.3: Compilación de lenguajes pasando por la generación de código intermedio.

En la Figura 6.2 y en la Figura 6.3 se observa cómo el número de traducciones y compilaciones se reduce cuando tenemos varios lenguajes fuente y varias máquinas destino existentes (en concreto, para n lenguajes y m plataformas, se reduce el número de traducciones para n y m mayores que dos).

El primer ejemplo práctico de esta técnica es el proyecto *UNCOL* (*Universal Computer Oriented Language*), que proponía un lenguaje intermedio universal para el diseño de compiladores [Steel60]. El objetivo de este proyecto era especificar una máquina abstracta universal para que los compiladores generasen código intermedio a una plataforma abierta. Otro ejemplo podemos encontrarlo en *ANDF* (*Architecture Neutral Distribution Format)* [Macrakis93] que tuvo como objetivo un híbrido entre la simplificación de compiladores y la portabilidad del código, aplicación que veremos en el siguiente punto. De esta forma, un compilador podría generar código para la especificación de la máquina *ANDF* siendo este código portable a distintas plataformas. Este código *ANDF* no era interpretado por un procesador *software*, sino que era traducido (o instalado) a código binario de una plataforma específica. Así se conseguía lo propuesto con *UNCOL*: la distribución de un código de una plataforma independiente.

Prácticas similares a las descritas también han sido adoptadas por varias compañías que desarrollan diversos tipos de compiladores. Un ejemplo son los productos de *Borland/Inprise* [Borland06]. Inicialmente esta compañía seleccionó un mismo "*back end*" para todos sus compiladores, especificando un formato binario para una máquina compartida por todas sus herramientas (archivos de extensión *obj*). Módulos de aplicaciones desarrolladas en distintos lenguajes como *C++* y *Delphi* podrían enlazarse para generar una aplicación, siempre que hayan sido compiladas a una misma plataforma [Trados96].

El siguiente paso en esta estrategia es la especificación de una plataforma o máquina abstracta independiente del sistema operativo, que permita desarrollar aplicaciones en distintos lenguajes y sistemas operativos y que también permita por tanto ampliar ese concepto de interoperabilidad entre lenguajes, entre otras ventajas. Este proyecto bautizado como "Proyecto *Kylix*" especifica un modelo de componentes *CLX* (*Component Library Cross-Platform*) que permite desarrollar aplicaciones en *C++ Builder* y *Delphi* para los sistemas operativos *Win32* y *Linux* [Kozak00].

6.3.1.1 ENTORNOS DE PROGRAMACIÓN MULTILENGUAJE

Apoyándose de forma directa en los conceptos de máquinas abstractas y máquinas virtuales, se han desarrollado entornos integrados de desarrollo de aplicaciones multilenguaje. *POPLOG* es un entorno de programación multilenguaje enfocado al desarrollo de aplicaciones de inteligencia artificial [Smith92]. Utiliza compiladores incrementales de *Common Lisp*, *Pop-11*, *Prolog* y *Standard ML*. La capacidad de interacción entre los lenguajes reside en la traducción a una máquina abstracta de alto nivel (*PVM*, *Poplog Virtual Machine*) y la independencia de la plataforma física utilizada se obtiene gracias a la compilación a una máquina de bajo nivel (*PIM*, *Poplog Implementation Machine*) y su posterior conversión a una plataforma física, como se veía en la Figura 6.3. El estándar *CLI* (cuya implementación más conocida es la máquina *.NET* de *Microsoft* [Microsoft05]) entra dentro de esta categoría, y al igual que el sistema mencionado anteriormente también es independiente de la plataforma, existiendo diferentes *ports* de la misma [MonoVSDotGNU05].

En este estándar, y en concreto en la máquina *.NET* de *Microsoft,* todos los lenguajes se ejecutan en una máquina virtual común llamada *CLR* (*Common Language Runtime*). Estos lenguajes poseen (entre otras propiedades) interoperabilidad, de manera que se puede usar cualquier componente desarrollado en un lenguaje desde el código de cualquier otro, gracias a que cualquier lenguaje implementado sobre dicha plataforma se compila a un lenguaje intermedio común o *CIL* (*Common Intermediate Language*). Esta capacidad también permite a los lenguajes implementados en este entorno tener acceso a la misma librería de clases base común o *BCL* (*Base Class Library*), proporcionándoles multitud de funcionalidades (comunicaciones, etc.) comunes a cualquiera de ellos. Esto permite que una vez que se sabe usar cierta librería con uno de los lenguajes, podemos usarla de la misma forma desde cualquiera de ellos. Podemos ver un esquema de esta plataforma en esta figura:

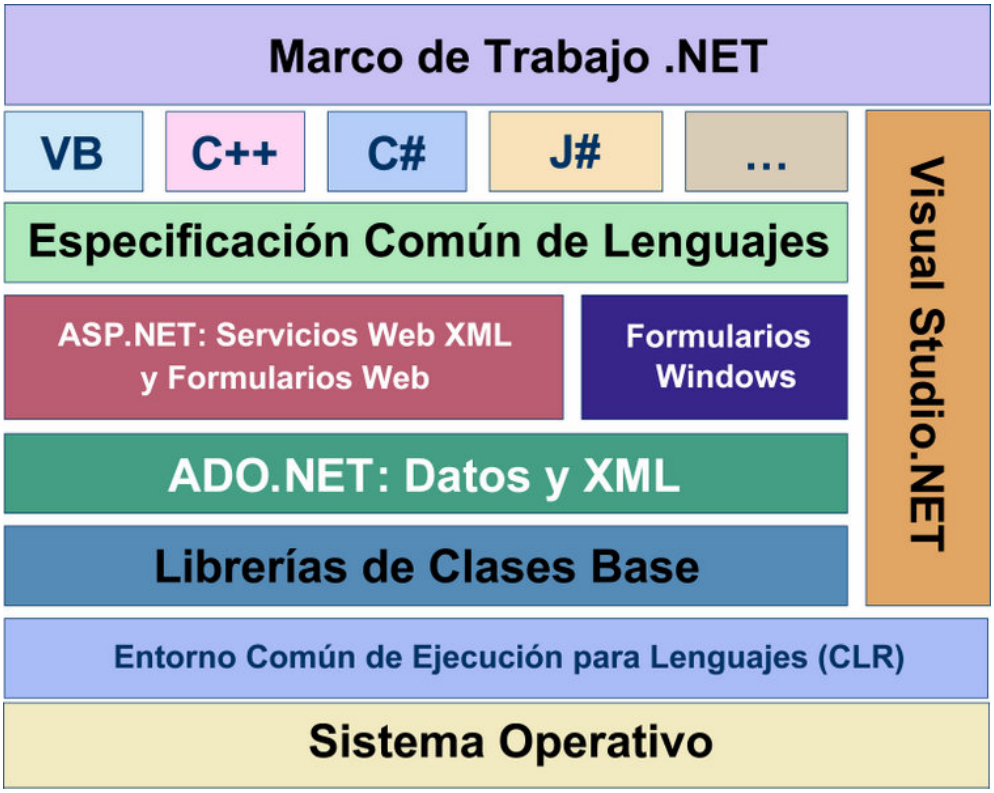

Figura 6.4: Esquema de la estructura del *CLR* [WikiDNET06]

6.3.2 Portabilidad del Código

Aunque todos los procesadores *hardware* son realmente implementaciones físicas de una máquina abstracta, como ya se ha dicho es común utilizar el concepto de máquina abstracta para designar la especificación de una plataforma cuyo objetivo final no es su implementación física, ya que el hecho de que sea un procesador *software* el que interpreta las instrucciones de la máquina da lugar a una independencia de la plataforma física real utilizada. Una vez codificado un programa para una máquina abstracta, su ejecución podrá realizarse en cualquier plataforma (microprocesador + sistema operativo) que posea un procesador lógico capaz de interpretar sus instrucciones, como se muestra en la siguiente figura:

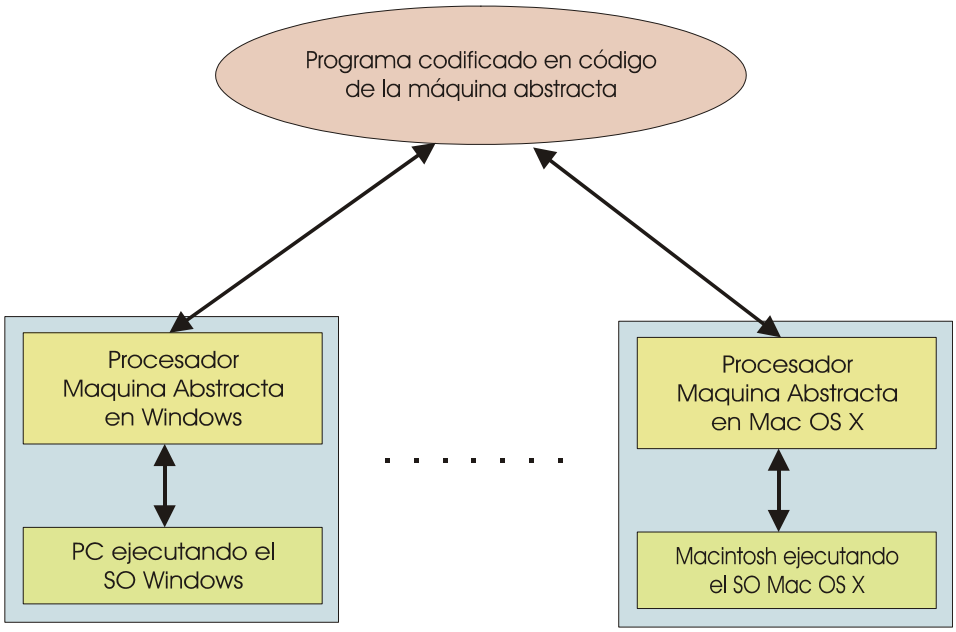

Figura 6.5: Ejecución de un programa portable sobre varias plataformas.

Al igual que un compilador de un lenguaje de alto nivel genera código para una máquina específica, la compilación a una máquina abstracta hace que ese programa generado sea independiente de la plataforma que lo ejecute, ya que dicho programa podrá ser ejecutado por cualquier procesador (*hardware* o *software*) que implemente la especificación computacional de un procesador de la máquina abstracta.

Cada procesador computacional de la máquina abstracta podrá estar implementado acorde a las necesidades y características del sistema real. En la Figura 6.6 se identifican distintos grados de implementación del procesador de la máquina. En el primer ejemplo se implementa la máquina físicamente obteniendo una mayor velocidad de ejecución del programa al tener un único nivel de interpretación. Si, en cambio, se implementase un procesador lógico sobre una plataforma distinta, obtenemos la portabilidad mencionada en este punto, pero perdiendo velocidad de ejecución frente al caso anterior. Además, cada plataforma física que emule la máquina abstracta implementará de forma distinta este procesador en función de sus recursos. En el ejemplo mostrado en la Figura 6.5, el procesador implementado sobre el *PC* con *Windows* podría haber sido desarrollado de forma distinta a la implementación sobre el *Macintosh* con *OS X*.

En la Figura 6.6 se muestra otro ejemplo en el que el emulador de la máquina abstracta es desarrollado sobre otro procesador lógico. Seguimos teniendo portabilidad

del código y obtenemos una flexibilidad en el propio emulador de la máquina abstracta. De esta forma, se gana en flexibilidad a costa de aumentar en número de computaciones necesarias en la ejecución de un programa de la máquina abstracta.

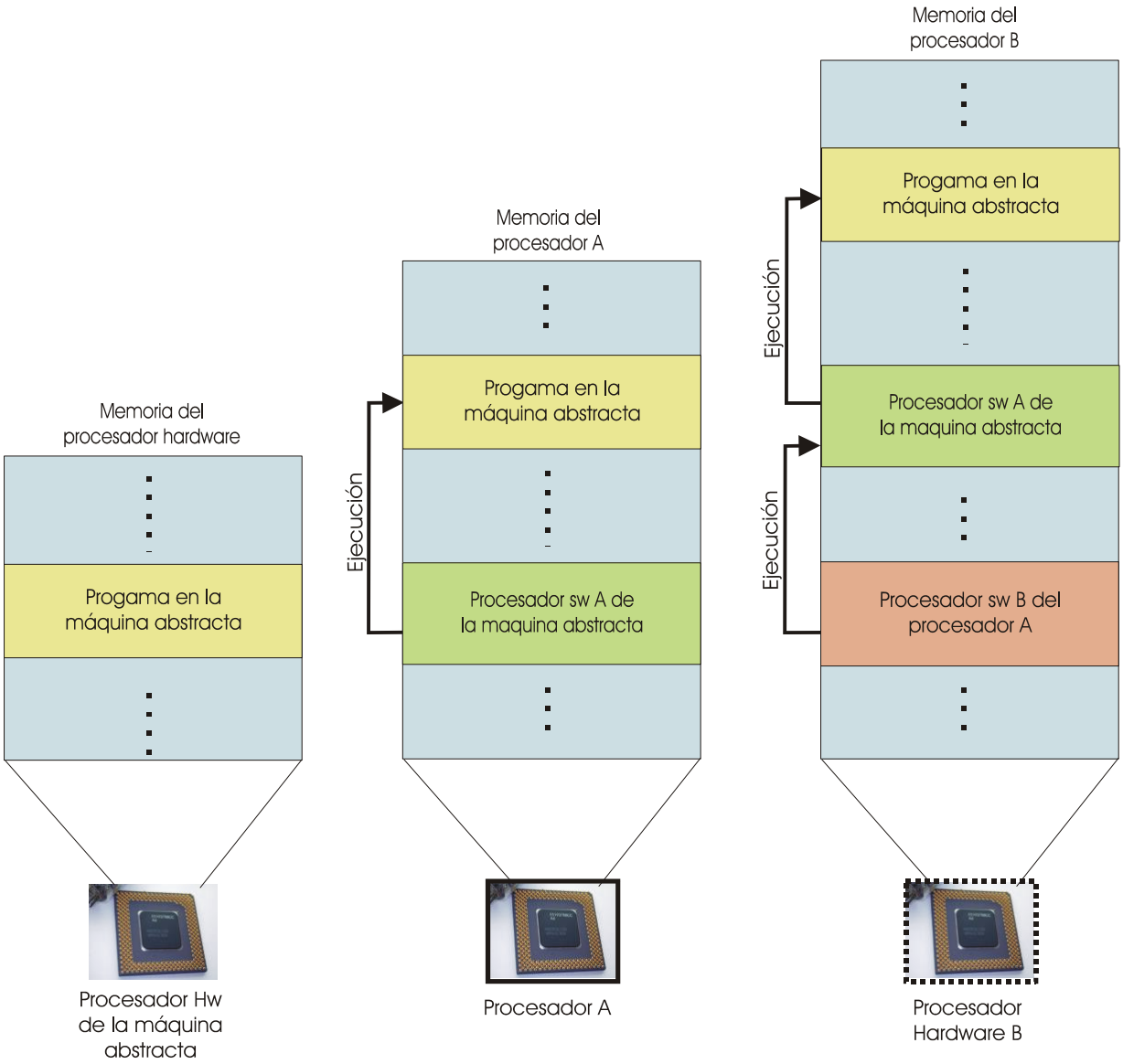

Figura 6.6: Distintos niveles de implementación de un procesador.

Existen multitud de casos prácticos que utilizan el concepto de máquina abstracta para conseguir programas portables a distintas plataformas. En la siguiente sección se verán un conjunto de éstos.

6.3.3 Sistemas Interactivos con Abstracciones Orientadas a Objetos

La utilización de los lenguajes orientados a objetos en la década de los 80 ha aumentado considerablemente el nivel de abstracción en la programación de aplicaciones

[Booch94]. Sin embargo, los microprocesadores existentes se seguían basando en la ejecución de código no estructurado, por lo que surgió una línea de desarrollo para crear microprocesadores que pudiesen operar con el nuevo paradigma nativamente. No obstante, todos los intentos de desarrollar procesadores con este "nuevo" paradigma finalizaron en fracaso debido a la falta de eficiencia obtenida causada por la complejidad de la implementación [Colwel88]. Posteriormente se llevaron a cabo estudios que concluyeron que, haciendo uso de optimizaciones de compilación e interpretación, se pueden obtener buenos rendimientos igualmente, y por tanto no es rentable el desarrollo de plataformas físicas orientadas a objetos [Hölzle95].

Para desarrollar entornos de programación y sistemas que aportasen directamente la abstracción de orientación a objetos totalmente interactivos, se utilizó el concepto de máquina abstracta orientada a objetos. Los sistemas, al igual que los lenguajes de programación, ofrecían la posibilidad de crear y manipular una serie de objetos que residían en la memoria de una máquina abstracta. Las diferencias entre trabajar con una máquina abstracta en lugar de compilar a la plataforma nativa son las propias de la dualidad compilador/intérprete: Un intérprete posee una menor eficiencia en ejecución y compilación, pero a cambio se obtiene una mayor portabilidad, flexibilidad y capacidad de interacción entre aplicaciones. Además, como ya se ha visto, utilizar un procesador lógico generaría un mayor número de computaciones en la interpretación, perdiendo rendimiento en ejecución pero ganando otras ventajas como la portabilidad de código. En cualquier caso, si la plataforma que interpreta el lenguaje es de un nivel más cercano al mismo (es orientada a objetos), los tiempos de compilación se reducirán al no haber cambio de paradigma, de ahí la utilización de máquinas abstractas orientadas objetos para este fin.

Como se muestra en Figura 6.7, el procesamiento lógico de los programas hace que todos ellos compartan una zona de memoria asociada a un proceso: El intérprete. La interacción entre aplicaciones es homogénea, y por tanto más sencilla de implementar que en el caso de que se cree un proceso distinto para la ejecución de cada aplicación. El hecho de que las aplicaciones puedan interactuar fácilmente y que se permita el acceso y modificación de los distintos objetos existentes aumenta la flexibilidad global del sistema, pudiendo representar sus funcionalidades mediante estos objetos y métodos modificables.

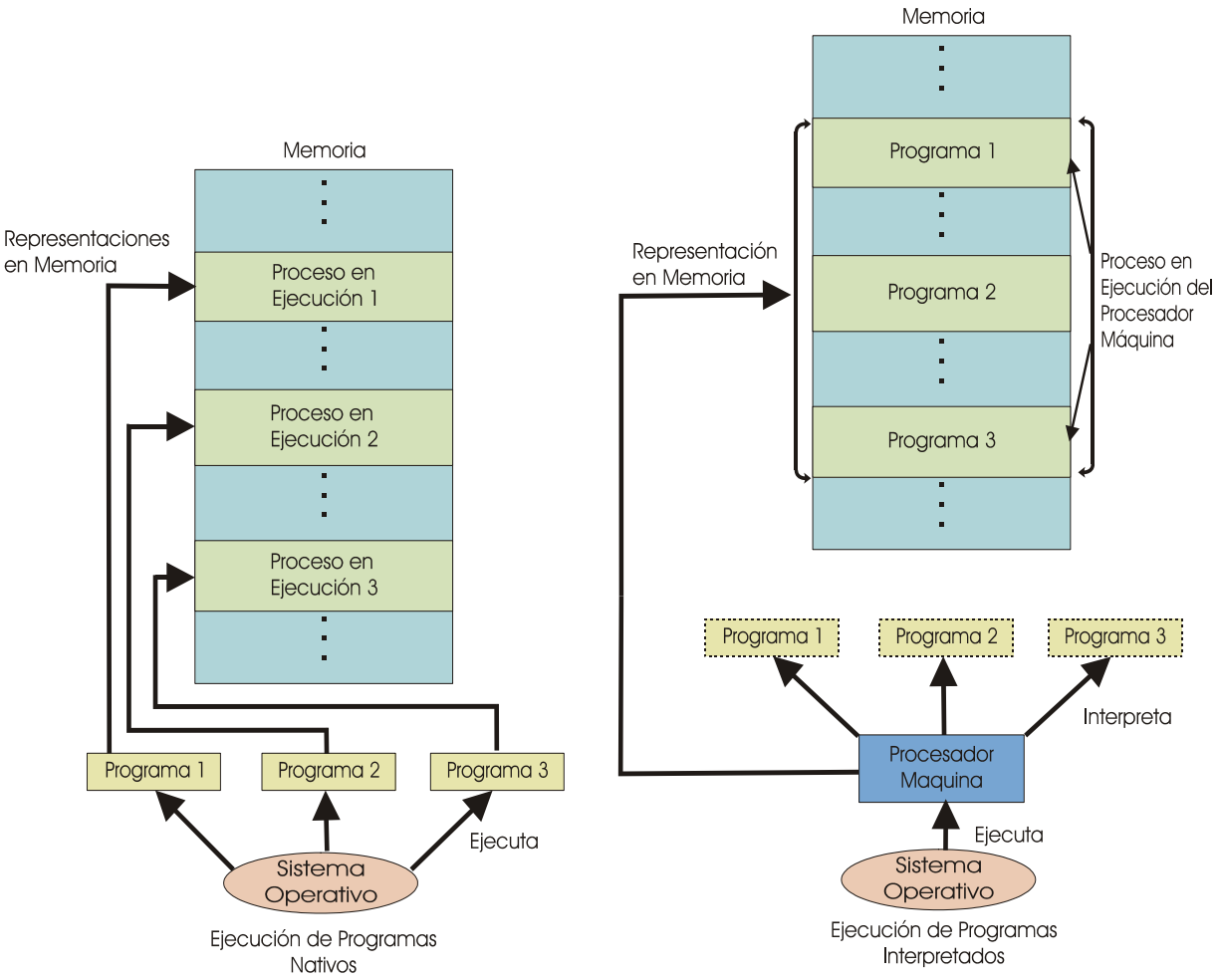

Figura 6.7: Diferencia entre la ejecución de programas nativos frente interpretados.

A modo de conclusión, podemos decir que los sistemas desarrollados sobre una máquina abstracta orientada a objetos facilitan al usuario:

- La utilización del sistema con una abstracción más natural a la forma de pensar del ser humano [Booch94].
- La autodocumentación del sistema y sus aplicaciones. El acceso a cualquier objeto permite conocer la estructura y comportamiento de éste en cualquier momento.
- La programación interactiva y continua. Una vez que el usuario entra en el sistema, accede a un "mundo" interactivo de objetos [Smith95], pudiendo hacer uso de cualquier objeto existente. Si se necesita desarrollar una nueva funcionalidad, se crearán nuevos objetos, definiendo su estructura y comportamiento, comprobando su correcta funcionalidad y utilizando cualquier otro objeto existente de una forma totalmente interactiva. A partir de ese momento el sistema poseerá una nueva funcionalidad y un mayor número de objetos.

Dentro de este tipo de sistemas se pueden nombrar a los clásicos *Smalltalk* [Mevel87] y *Self* [Ungar87], ya vistos anteriormente.

6.3.4 Distribución e Interoperabilidad de Aplicaciones

Como ya se ha dicho, el hecho de tener una aplicación codificada sobre una máquina abstracta implica que ésta podrá ejecutarse en cualquier plataforma que implemente esta máquina virtual. Esto puede ofrecer dos ventajas adicionales:

- Una aplicación podrá ser distribuida a lo largo de una red de computadores. Los distintos módulos codificados para la máquina abstracta pueden descargarse y ejecutarse en cualquier plataforma física que implemente la máquina virtual.
- Las aplicaciones podrán interoperar entre sí (envío y recepción de datos) de forma independiente a la plataforma física sobre la que estén ejecutándose. La representación de la información nativa de la máquina abstracta será interpretada por la máquina virtual de cada plataforma.

Aunque de los beneficios derivados de la utilización de una máquina abstracta ya estaban presentes en *Smalltalk* [Goldberg83] y *Self* [Smith95], aunque su mayor auge tuvo lugar con la aparición de las plataformas virtuales *Java* [Kramer96] y *.NET* [Microsoft05], que han impulsado el desarrollo de aplicaciones distribuidas, especialmente a través de *Internet*.

6.3.4.1 DISTRIBUCIÓN DE APLICACIONES

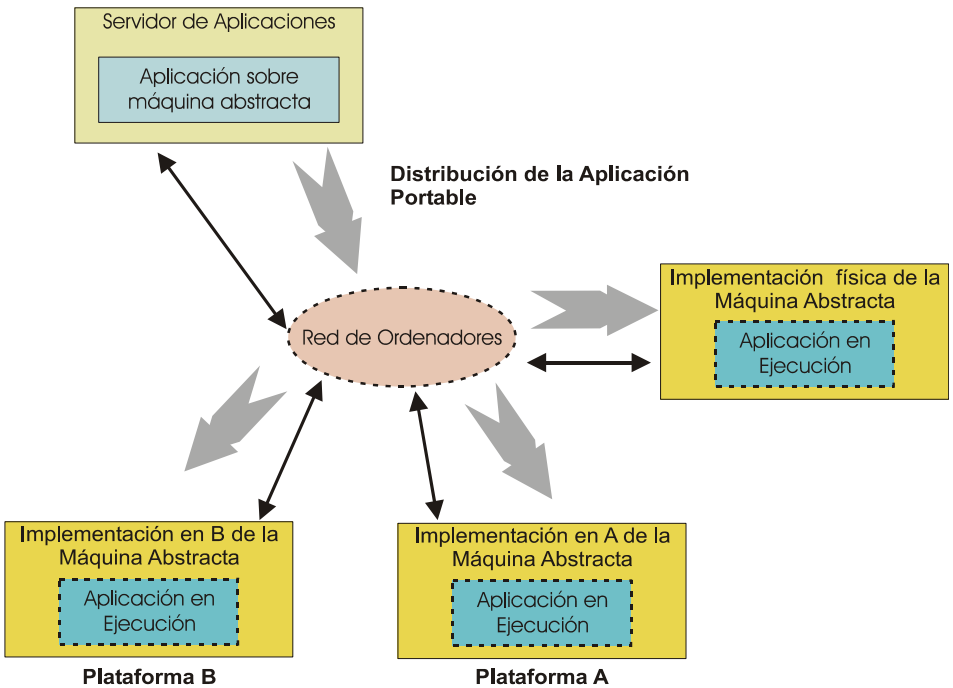

Figura 6.8: Distribución de aplicaciones portables

En la Figura 6.8 se muestra un escenario de distribución de una aplicación desarrollada sobre una máquina abstracta. Un ordenador servidor de la aplicación,

conectado mediante una red de ordenadores a un conjunto de clientes, posee el código del programa a distribuir. Mediante un determinado protocolo de comunicaciones, cada cliente demanda la aplicación al servidor, la obtiene en su ubicación y la ejecuta en su implementación de la máquina virtual. Esta ejecución será computacionalmente similar en todos los clientes, con el aspecto propio de la plataforma física en la que se interpreta.

Un caso de uso típico del escenario mostrado es la descarga de *applets* en *Internet*. El servidor de aplicaciones es un servidor *Web,* mientras que el código de la aplicación es un código *Java* con restricciones denominado *applet* [Kramer96]. El cliente, mediante su navegador *Web*, se conecta al servidor utilizando el protocolo *HTTP* [Beners96] o *HTTPS* [Freier96] y descarga el código *Java* en su navegador. La aplicación es interpretada por la máquina virtual de *Java* [Sun95] implementada en el navegador, de forma independiente a la plataforma y navegador que el cliente use.

6.3.4.2 INTEROPERABILIDAD DE APLICACIONES

Uno de los mayores problemas en la intercomunicación de aplicaciones distribuidas físicamente es la representación de la información enviada. Si desarrollamos dos aplicaciones nativas sobre dos plataformas distintas y hacemos que intercambien información, deberemos preestablecer la representación de la información utilizada. Por ejemplo, una variable entera en el lenguaje de programación *C* [Ritchie78] puede tener una longitud y representación binaria distinta en cada una de las plataformas (como por ejemplo podría ocurrir entre plataformas *big* y *little endian*), por lo que es necesario un "acuerdo" que haga a la información enviada legible por cualquier plataforma.

Además de definir la representación de la información y traducir ésta a su representación nativa, también se debe definir el protocolo de comunicación, es decir, el modo en el que las aplicaciones deben dialogar para intercambiar dicha información. Existen especificaciones estándar definidas para interconectar aplicaciones nativas sobre distintas plataformas. Un ejemplo es el complejo protocolo *GIOP* (*General Inter-ORB Protocol*) [OMG95] definido en *CORBA* [Baker97], que establece el protocolo y la representación de información necesaria para interconectar cualquier aplicación nativa a través de la arquitectura de objetos distribuidos *CORBA*. Este tipo de *middleware* proporciona un elevado nivel de abstracción, facilitando el desarrollo de aplicaciones distribuidas, pero no está exento de inconvenientes:

- Requiere una elevada cantidad de código adicional para implementar el protocolo y la traducción de la información enviada por la red. Este código recibe el nombre de *ORB* (*Object Request Broker*).
- Aumenta el volumen de información enviada a través de la red, al implementar un protocolo de propósito general que proporcione un mayor nivel de abstracción.

Si las aplicaciones se desarrollan sobre una misma máquina abstracta, el envío de la información se puede realizar directamente en el formato nativo de ésta puesto que existe una máquina virtual en toda plataforma. La traducción de la información de la máquina a la plataforma física la lleva a cabo en el intérprete de la máquina virtual. El resultado es poder interconectar aplicaciones codificadas en una misma máquina abstracta y ejecutadas en diferentes plataformas o dispositivos totalmente dispares, con lo que se logra que el mismo programa pueda funcionar y/o comunicarse entre plataformas arquitectónicamente muy diferentes, como ocurre hoy en día con la telefonía móvil y las aplicaciones *Java*.

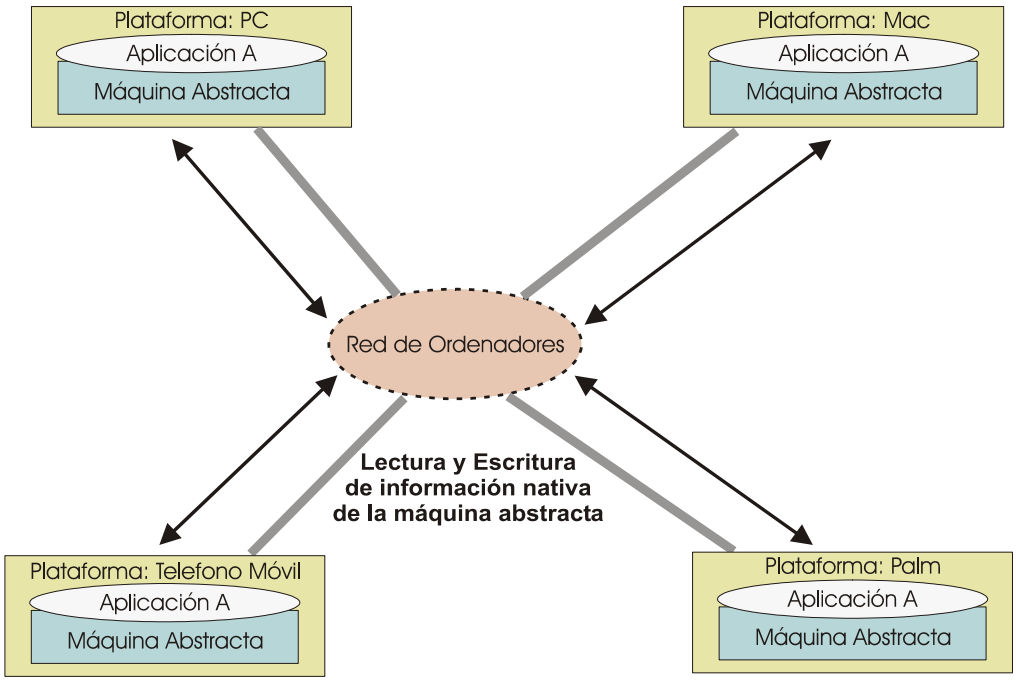

Figura 6.9: Interoperabilidad nativa de aplicaciones sobre distintas plataformas.

En el lenguaje de programación *Java* [Gosling96] es posible enviar los datos en su representación nativa. Además, se puede enviar cualquier tipo de dato, no sólo tipos simples, ya que los objetos son enviados convirtiéndolos a una secuencia de *bytes*, proceso que se denomina serialización (*serialization*) [Campione99]. Además, mediante el paquete de clases ofrecidas en `java.net`, es posible implementar un protocolo propio de un tipo de aplicaciones sobre *TCP/IP* o *UDP/IP* [Raman98]. Si deseamos obtener un mayor nivel de abstracción para el desarrollo de aplicaciones distribuidas, al igual que teníamos con *CORBA*, podemos utilizar *RMI* (*Remote Method Invocation*) sin sobrecargar tanto la red de comunicaciones [Sun97e]. *RMI* desarrolla un protocolo de interconexión de aplicaciones *Java* llamado *JRMP* (*Java Remote Method Protocol*) que permite, entre otras cosas, invocar a métodos de objetos ubicados en otras máquinas virtuales. Existen otros mecanismos similares sobre la plataforma *.NET* (*.NET Remoting* [Srinivasan03]).

6.3.5 Diseño y Coexistencia de Sistemas Operativos

La utilización del concepto de máquina abstracta ha estado presente también en el desarrollo de sistemas operativos. Debido a que su función está fuera del ámbito de esta tesis, sólo se hará aquí una breve reseña de sus posibles aplicaciones. Podemos clasificar su utilización en función del objetivo buscado de la siguiente forma:

- **Desarrollo de sistemas operativos distribuidos y multiplataforma**.
- **Ejecución de aplicaciones desarrolladas sobre cualquier sistema operativo**: Existen sistemas operativos, como el *VM/ESA* de *IBM* [IBM00], que utilizan la conjunción de ambas funcionalidades en la utilización de una máquina abstracta.

- **Despliegue rápido de aplicaciones**: Máquinas virtuales con un conjunto de programas inmediatamente preparados para su funcionamiento.
- **Aislamiento de información**: Permitir separar totalmente programas que se ejecutan sobre una misma máquina.

6.3.5.1 DISEÑO DE SISTEMAS OPERATIVOS DISTRIBUIDOS Y MULTIPLATAFORMA

Estos sistemas operativos aprovechan todas las ventajas de la utilización de máquinas abstractas comentadas en los puntos anteriores para desarrollar un sistema operativo distribuido y multiplataforma. Sobre la descripción de una máquina abstracta, se implementa un intérprete de la máquina virtual en toda aquella plataforma en la que vaya a desarrollarse el sistema operativo. En el código de la máquina se desarrollarán servicios propios del sistema operativo que permitan interactuar con el sistema y elevar el nivel de abstracción con los siguientes objetivos:

- Una aplicación para este sistema operativo se puede portar a cualquier plataforma.
- Las aplicaciones no se limitan a utilizar los servicios de la máquina, sino que podrán codificarse en un mayor nivel de abstracción: el ofrecido por los servicios sistema operativo.
- El propio sistema operativo es portable, puesto que ha sido desarrollado sobre la máquina abstracta. No es necesario pues, codificar cada servicio para cada plataforma, sino sólo la parte correspondiente a la máquina virtual.
- La interoperabilidad de las aplicaciones es uniforme, al estar utilizando únicamente el modelo de computación de la máquina abstracta. En otros sistemas operativos es necesario especificar la interfaz exacta de acceso a sus servicios.
- Las aplicaciones desarrolladas sobre este sistema operativo pueden ser distribuidas físicamente por el sistema operativo, ya que todas serán ejecutadas por un intérprete de la misma máquina abstracta.
- En la comunicación de aplicaciones ejecutándose en computadores distribuidos físicamente no será necesario establecer traducciones de datos. La interacción es directa, al ejecutarse todas las aplicaciones sobre la misma máquina abstracta pero en distintas máquinas virtuales o intérpretes.

Existen distintos sistemas operativos desarrollados sobre una máquina abstracta, ya sean comerciales, de investigación o didácticos, pero en esta tesis no se hará una descripción de los mismos al estar fuera de los objetivos pretendidos en la misma.

6.3.5.2 COEXISTENCIA DE SISTEMAS OPERATIVOS

Este concepto puede definirse como el acceso uniforme a los recursos de una plataforma física (de una máquina real). Es una interfaz de interacción con una plataforma física que puede ser dividida en un conjunto de máquinas virtuales. La partición de los recursos físicos de una plataforma, mediante un acceso independiente, permite la ejecución de distintos sistemas operativos dentro del operativo que se encuentre en ejecución. Dentro de un sistema operativo se desarrollarían tantas máquinas virtuales como sistemas inmersos deseemos tener. La ejecución de una

aplicación desarrollada para un sistema operativo distinto al activo se producirá sobre la máquina virtual implementada para el sistema operativo "huésped". Esta ejecución utilizará los recursos asignados a su máquina virtual.

La ventaja obtenida se resume en la posibilidad de ejecutar aplicaciones desarrolladas sobre cualquier sistema operativo sin tener que reiniciar el sistema, es decir, sin necesidad de cambiar el sistema operativo existente en memoria. No obstante, el principal inconveniente es la carencia de interacción entre las aplicaciones ejecutadas sobre distintos sistemas operativos: no es posible comunicar las aplicaciones "huésped" entre sí, ni con las aplicaciones del propio operativo.

El primer sistema operativo que utilizó de esta forma una máquina virtual fue el *OS/2* de *IBM*. Se declaró como el sucesor de *IBM* del sistema operativo *DOS*, desarrollado para microprocesadores *Intel 80286*. Este sistema era capaz de ejecutar en un microprocesador *Intel 80386* varias aplicaciones *MS-DOS*, *Windows* y aplicaciones gráficas nativas, haciendo uso de la implementación de distintas máquinas virtuales. *IBM* abandonó el proyecto iniciado con su sistema operativo *OS/2*, pero parte de sus características fueron adoptadas en el desarrollo de su operativo *VM/ESA,* desarrollado para servidores *IBM S/390* [IBM00]. Este sistema permite ejecutar aplicaciones desarrolladas para otros sistemas operativos, utilizando una máquina virtual como modo de acceso a los recursos físicos. El empleo de una máquina virtual es aprovechado además para la portabilidad e interoperabilidad del código: cualquier grupo de aplicaciones desarrolladas sobre esta máquina abstracta puede interoperar entre sí, de forma independiente al tipo de servidor sobre el que se estén ejecutando. Otro producto que utiliza el concepto de máquina virtual para multiplexar el acceso a una plataforma física es *VMware Virtual Platform* [Jones99]. Ejecutándose en *Windows NT/2k/XP* o en *Linux* permite lanzar aplicaciones codificadas para *Windows 3.1*, *Windows 9X*, *MS-DOS*, *Windows NT/2K/XP*, *Linux*, *FreeBDS* y *Solaris 7* para *Intel*.

6.3.5.3 DESPLIEGUE RÁPIDO DE APLICACIONES

Es posible tener máquinas virtuales preconfiguradas que contengan un conjunto de aplicaciones determinado completamente listo para su funcionamiento, de manera que se puedan usar para un rápido despliegue de aplicaciones. Las máquinas virtuales pueden arrancarse sin tareas innecesarias que consuman tiempo, como instalar y configurar *software* del que dependa la aplicación. Este mecanismo de despliegue puede ser muy útil para la demostración de funcionamiento de entornos *software* complejos (aplicaciones cliente-servidor que puedan requerir múltiples máquinas, una aplicación *Web* típica, etc.) [Rosenblum04].

6.3.5.4 AISLAMIENTO DE INFORMACIÓN

La capacidad de aislar la información manipulada por máquinas virtuales que se implementan a nivel de *hardware* (como las funcionalidades soportadas al respecto por muchos microprocesadores modernos) ha sido usada para aspectos de seguridad.

Un ejemplo es el sistema *NetTop* de la *NSA* (*National Security Agency*) [NSA06], que usa esta capacidad para que una misma estación de trabajo pueda acceder a una red de información clasificada y a la red pública *Internet*. Para ello se usan dos máquinas virtuales, una de ellas accede sólo a la red clasificada y la otra hace lo mismo con la pública. De esta forma ningún atacante proveniente de la red pública podrá acceder a la información clasificada [Rosenblum04].

Este sistema permite hacer funcionar sistemas *MSL* (*Multiple Single Level*) sobre

un sistema operativo *Linux* con características de seguridad mejoradas, que posee varias instancias del programa *VMWare* en funcionamiento, con *Windows* como sistema operativo cliente. Las aplicaciones *MSL* son un método de separación de diferentes niveles de datos sometidos a requisitos de seguridad más o menos restrictivos, usando diferentes *PCs* o máquinas virtuales para cada nivel. Esto permite tener los beneficios de la seguridad multinivel (que se refiere a la capacidad de un sistema para tratar información con diferente nivel de seguridad adecuadamente) sin necesidad de hacer cambios ni a las aplicaciones ni a los sistemas operativos, pero con el coste de adquirir *hardware* adicional o más potente.

6.3.6 Protección de Derechos de Autor

La industria del *software* ha padecido desde siempre el mal de la piratería, que recientemente ha llegado a cotas muy elevadas. La copia masiva e indiscriminada de *software* y contenidos multimedia por parte de casi cualquier usuario particular, y la distribución de contenidos de este tipo de forma libre por la red han ocasionado muchas pérdidas a esta industria, que amenazan seriamente su continuidad en algunos campos. Es por ello que los fabricantes y distribuidores usan sistemas creados con el propósito de imposibilitar o dificultar enormemente la copia de estos materiales, con el objetivo de limitar en la medida de lo posible la distribución de copias del mismo. Aunque en algunos casos estos sistemas anticopia no están exentos de polémicas, por los presuntos perjuicios causados a aquellos usuarios que sí compran el producto original, en este campo existen sistemas desarrollados empleando máquinas virtuales con un grado de eficacia muy elevado, aunque no existe aún un sistema anticopia totalmente infalible.

Estos sistemas que emplean máquinas virtuales para proteger una aplicación específica de copias no autorizadas normalmente se basan en técnicas como, por ejemplo, *Code Morphing* (transformación de código), que permite transformar instrucciones de código intermedio, como las resultantes de compilar programas *Java* o de la plataforma *.NET*, en otras instrucciones que pueden ser de una máquina virtual creada al efecto, a través de un conjunto muy grande de patrones de transformación, o bien convertir código binario un formato que no sea descifrable muy diferente del código binario normal. El código transformado de esta forma se ejecuta en su forma ya convertida, ya que normalmente no es posible "restaurar" el código "original".

Entre estos sistemas que usan máquinas virtuales describiremos *Starforce*, un sistema usado para la protección de *software* comercial (principalmente videojuegos), de un alto nivel de efectividad, y el nuevo sistema de protección contra copias de los discos *Blu-Ray*.

6.4 PANORÁMICA DE UTILIZACIÓN DE MÁQUINAS ABSTRACTAS

Una vez vistos los conceptos principales y las aplicaciones de las máquinas abstractas, a continuación se hará un estudio de los sistemas basados en las mismas, con el objeto de ver que características poseen.

6.4.1 Máquinas que Facilitan la Portabilidad de Código

Como ya se ha comentado, una de las ventajas más explotada en la utilización de máquinas abstractas es la portabilidad de su código. Al no ser necesario implementar físicamente su procesador computacional, entonces el código desarrollado para esta plataforma puede ser ejecutado en cualquier sistema que implemente dicho procesador lógico, que al ser *software* puede migrarse mucho más fácilmente a diferentes plataformas. A continuación estudiaremos un conjunto de casos prácticos en los que se utilizan máquinas abstractas para obtener, principalmente, la portabilidad de un determinado código generado.

6.4.1.1 MÁQUINA-P

El *código-p* es el lenguaje intermedio propio de una máquina abstracta llamada *máquina-p* [Nori76], utilizada inicialmente en el desarrollo de un compilador del lenguaje *Pascal* [Jensen91]. La Universidad de California en San Diego (*UCSD*) desarrolló un procesador que ejecutaba código binario de la *máquina-p* (*código-p*) y se adoptó la especificación de la máquina abstracta para desarrollar así un proyecto de *Pascal* portable. Se llegó a disponer de soporte para multitarea e incluso se desarrolló el *p-System*: un sistema operativo portable, codificado en *Pascal* y traducido a *código-p* [Campbell83]. Por tanto, la única parte del sistema que se debía implementar en una plataforma concreta para portar este sistema era el emulador de la *máquina-p*. El sistema se llegó a implantar en diversos microprocesadores como *DEC LSI-11*, *Zilog Z80*, *Motorola 68000* e *Intel 8088* [Irvine99].

Al igual que los lenguajes *C* y *Algol*, el *Pascal* es un lenguaje orientado a bloques u orientado a marcos de pila (por cada bloque se apila un marco o contexto propio de la ejecución de ese bloque) si lo vemos desde el punto de vista de su implementación [Jensen91]. Esto determino que la especificación de la *máquina-p* fuese orientada a una estructura de pila. El *p-System* implementado tenía la siguiente estructura en memoria, desde las direcciones superiores a las inferiores:

- El código (*p-código*) propio del sistema operativo (*p-System*).
- La pila del sistema (creciendo en sentido descendente).
- La memoria *heap* (creciendo en sentido ascendente).
- El conjunto de las pilas propias de sus hilos según se iban demandando en tiempo de ejecución.
- Los segmentos globales de datos (de constantes y variables).
- El intérprete o simulador de la máquina abstracta.

A diferencia de otras máquinas, esta máquina abstracta sí llegó a tener un procesador *hardware*. *Western Digital* implementó en 1980 la *máquina-p* en el *WD9000 Pascal Microengine,* basado en el microprocesador programable *WD MCP-1600*.

6.4.1.2 OCODE

OCODE [Richards71] fue el nombre asignado al lenguaje ensamblador de una máquina abstracta diseñada como una máquina de pila. Este lenguaje se utilizaba como código intermedio de un compilador de *BCPL* [Richards79], obteniendo la portabilidad del código generado entre los distintos sistemas soportados. *BCPL* (*Basic CPL*) es un lenguaje de sistemas desarrollado en 1969, descendiente de *CPL* (*Combined Prog. Language*). Este lenguaje se programa en un bajo nivel de abstracción, carece de tipos, está orientado a bloques y proporciona vectores de una dimensión y punteros, además de ser un lenguaje estructurado y poseer procedimientos con paso por valor.

En lo referente a la ejecución de aplicaciones, esta máquina posee una serie de mecanismos de comunicación mediante el uso de un sistema de memoria compartida, que permite almacenar variables del sistema y de usuario. A pesar de las limitaciones impuestas por su arquitectura, *BCPL* fue utilizado para desarrollar el sistema operativo *TRIPOS*, posteriormente renombrado a *AmigaDOS*.

6.4.1.3 PORTABLE SCHEME INTERPRETER

Bajo esta denominación se implementó un compilador del lenguaje *Scheme* [Abelson00] en una máquina virtual llamada *PSI* (*Portable Scheme Interpreter*). La compilación del código *Scheme* a la máquina virtual permite ejecutar la aplicación en cualquier sistema que posea este intérprete. Entre otras características, este sistema permitía añadir primitivas y depurar una aplicación con el *Portable Scheme Debugger*.

6.4.1.4 FORTH

Éste es otro ejemplo de especificación de una máquina abstracta cuyo objetivo era conseguir la portabilidad de código, en este caso del lenguaje *Forth,* mediante el uso de una máquina virtual [Brodie87]. Este lenguaje fue desarrollado en la década de los 70 por Charles Moore para el control de telescopios. Es un lenguaje sencillo, rápido y ampliable que es interpretado en una máquina virtual, consiguiendo ser portable a distintas plataformas y útil para empotrarlo en sistemas.

El simulador de la máquina virtual de *Forth* poseía dos pilas. La primera es la pila de datos: los parámetros de una operación son tomados del tope de la pila y el resultado es posteriormente apilado. La segunda pila es la pila de valores de retorno: se apilan los valores del contador de programa antes de una invocación a una subrutina, para poder retornar al punto de ejecución original una vez finalizada ésta.

6.4.1.5 SEQUENTIAL PARLOG MACHINE

En este caso, además de para facilitar la portabilidad del código, el concepto de máquina abstracta se utilizó para desarrollar un lenguaje multitarea. *SPM* (*Sequential Parlog Machine*) [Gregory87] es una máquina virtual del lenguaje de programación lógica *Parlog* [Clark83], también llamado "*Parallel-Prolog*".

6.4.1.6 *CODE WAR*

En este sistema se utiliza una máquina abstracta para hacer código portable, pero en este caso concreto la aplicación del mismo es un tanto peculiar: La portabilidad del código es aprovechada para crear programas que "luchen" entre sí, tratando de eliminarse los unos a los otros. *Code War* es un juego entre dos o más programas (no usuarios o jugadores) desarrollados en un lenguaje ensamblador denominado *Redcode*, que es el código nativo de una máquina abstracta denominada *MARS* (*Memory Array Redcode Simulator*) [Dewdney88]. El objetivo del juego es desarrollar un programa que sea capaz de eliminar todos los procesos de los programas contrarios que estuvieren ejecutándose en la máquina virtual, quedando tan sólo él en la memoria de la máquina.

Gracias a la utilización de una máquina virtual es posible desarrollar programas y jugar en *Code War* en multitud de plataformas: *UNIX*, *IBM PC compatible*, *Macintosh* y *Amiga*. Para tener una plataforma estándar de ejecución, se creó *ICWS* (*International Code War Society*), responsable de la creación y mantenimiento del estándar de la plataforma de *Code War*, así de cómo la organización de campeonatos.

El sistema en el que los programas son ejecutados es simple. El núcleo del sistema es su memoria: un vector de instrucciones inicialmente vacío. El código de cada programa es interpretado de forma circular, de manera que cuando finaliza su última instrucción, se vuelve a ejecutar la primera. La máquina virtual de *MARS* ejecuta una instrucción de cada programa en cada turno. Una vez evaluada la instrucción de un programa, se toma otro código y ejecuta una instrucción de éste. La interpretación de cada instrucción lleva siempre el mismo tiempo, un ciclo de la máquina virtual, sea cual sea su semántica. De esta forma, el tiempo de procesamiento es distribuido equitativamente a lo largo de todos los programas que estuvieren en memoria [Dewdney90], sin que haya programas que tengan más ventaja que otros. Cada programa podrá tener un conjunto de procesos en ejecución, almacenados por la máquina virtual en una pila de tareas. Cuando se produce el turno de ejecución de un programa, un proceso de éste se desapila y se ejecuta su siguiente instrucción. Si un proceso no se ha destruido durante la evaluación de su instrucción, es introducido nuevamente en la pila de tareas, continuando este proceso.

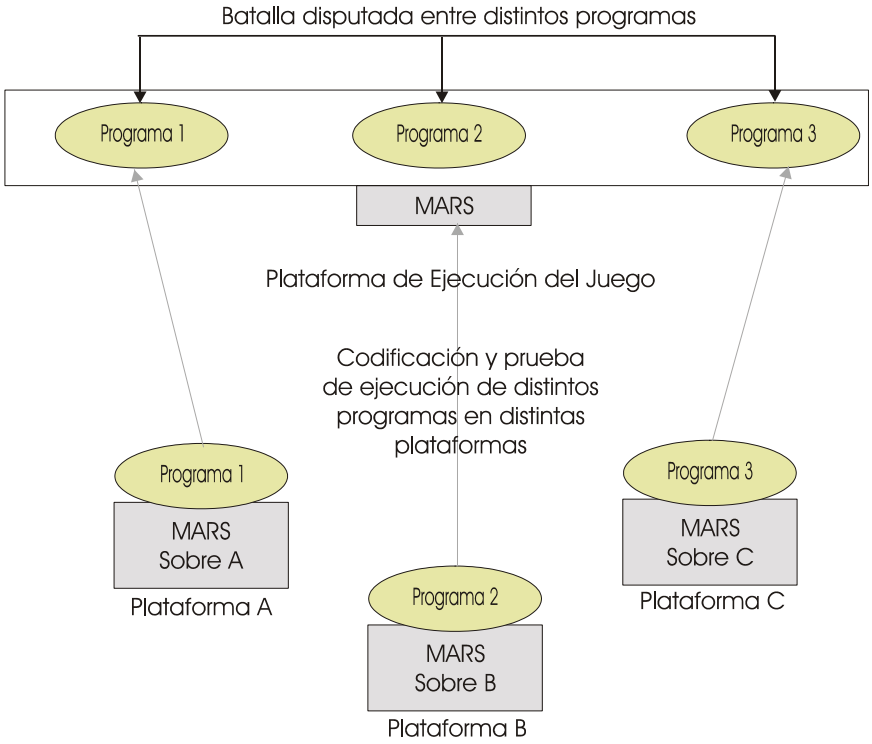

Figura 6.10: Desarrollo y ejecución de programas para *Code War*

6.4.1.7 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Todos los sistemas estudiados en esta sección están basados principalmente en la portabilidad del código escrito para una plataforma virtual. En la mayoría de los casos, los distintos lenguajes de programación son compilados a una plataforma intermedia y ésta es interpretada en distintas plataformas, dando lugar a la portabilidad del código generado. En el caso de *Code War*, esta característica supone la posibilidad de jugar con un programa desarrollado y probado en cualquier entorno.

Por otra parte, estos sistemas poseen una serie de limitaciones. Por ejemplo, todos los sistemas estudiados tienen una dependencia del lenguaje de programación que se desea que sea portable, diseñando la máquina abstracta en función de un lenguaje de programación de alto nivel. No obstante, la principal carencia de estas plataformas es que son diseñadas para resolver únicamente el problema de hacer que el código de un lenguaje sea portable. Es más, en el caso de *Code War*, la plataforma es desarrollada para permitir establecer batallas entre programas, que es una utilidad muy específica y de muy poca utilidad práctica para el desarrollo de aplicaciones reales.

6.4.2 Máquinas que Facilitan la Interoperabilidad entre Aplicaciones

Incluiremos en esta sección una serie de sistemas que usan el concepto de máquina abstracta como un medio para interconectar diferentes aplicaciones.

6.4.2.1 PARALLEL VIRTUAL MACHINE

Parallel Virtual Machine (*PVM*) permite la interacción de un conjunto heterogéneo de computadores *UNIX* interconectados entre sí, de manera que el conjunto pueda ser visto como una única máquina multiprocesador sobre la que corre un programa [Geist94]. Por tanto, *PVM* fue diseñado para interconectar los recursos de distintos computadores, proporcionando al usuario la abstracción de una plataforma multiprocesador capaz de ejecutar sus aplicaciones de forma independiente al número y ubicación de ordenadores usados.

El proyecto *PVM* comenzó en 1989 de forma no pública, aunque a partir de la versión 3 (1993) fue puesta en *Internet* para su descarga de manera gratuita, utilizándose en multitud de aplicaciones científicas. *PVM* está formado por un conjunto integrado de herramientas y librerías *software*, que emulan un entorno de programación de propósito general, flexible, homogéneo y concurrente, desarrollado sobre un conjunto heterogéneo de computadores interconectados. Los principios sobre los que se ha desarrollado, son:

- Acceso a múltiples computadores: Las tareas computacionales de una aplicación son ejecutadas en un conjunto de ordenadores. Este grupo de computadores es seleccionado por el usuario, de forma previa a la ejecución de un programa.
- Independencia del *hardware*: Los programas son ejecutados sobre un entorno de procesamiento virtual. Si el usuario desea ejecutar una determinada computación

sobre una plataforma concreta, podrá asociarla a una máquina en particular.

- Independencia del lenguaje: Los servicios de *PVM* han sido desarrollados como librerías estáticas para diversos lenguajes como *C* [Ritchie78], *C++* [Stroustrup98] o *Fortran* [Koelbel94].
- Computación basada en tareas: La unidad de paralelismo en *PVM* es la tarea, que es un hilo de ejecución secuencial e independiente. No existe una traducción directa de tarea a procesador, un procesador puede ejecutar múltiples tareas.
- Modelo de paso de mensajes explícitos: Las distintas tareas en ejecución realizan una parte del trabajo global, y todas ellas se comunican entre sí mediante el envío de mensajes explícitos.
- Entorno de computación heterogéneo: *PVM* soporta heterogeneidad respecto al *hardware*, redes de comunicación y aplicaciones.
- Soporte multiprocesador: Las plataformas multiprocesador pueden desarrollar su propia implementación de la interfaz *PVM*, obteniendo mayor eficiencia de un modo estándar.

La arquitectura de *PVM* aparece representada en la siguiente figura:

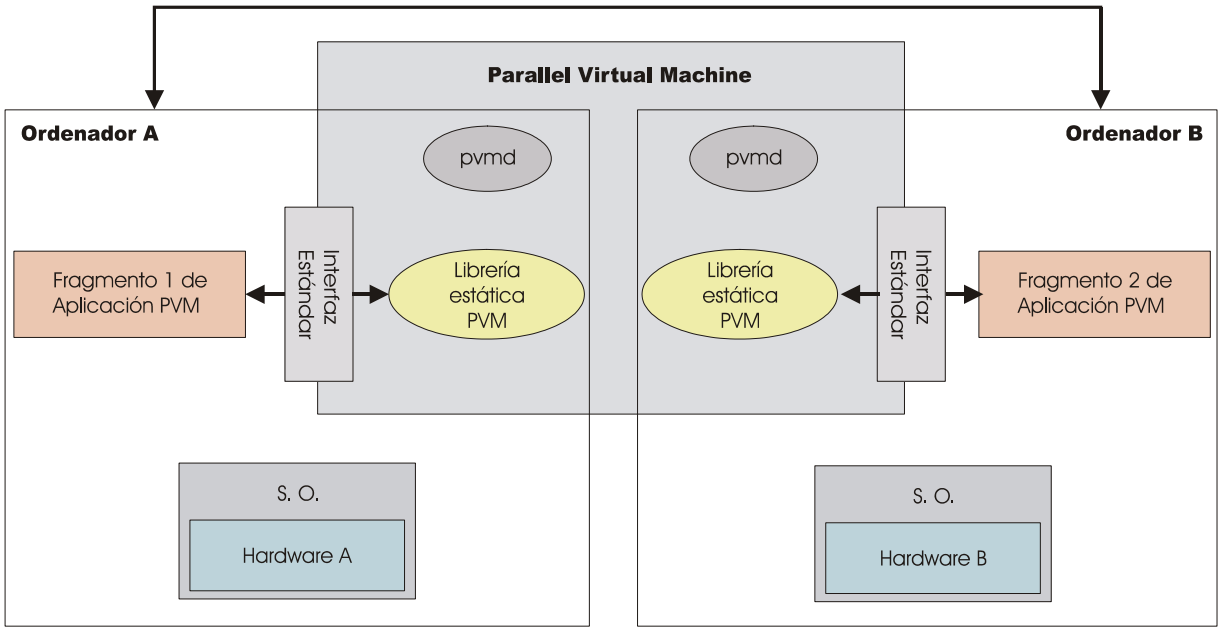

Figura 6.11: Arquitectura de *PVM*

Cada plataforma *UNIX* que desee utilizar *PVM* deberá ejecutar un proceso demonio (*daemon*) que será la máquina virtual. Eligiendo un lenguaje de programación, el acceso al sistema heterogéneo y distribuido se realiza mediante una interfaz de invocación a una librería estática. Esta librería ofrece un conjunto de rutinas necesarias para la intercomunicación de las distintas tareas de una aplicación. La implementación de dichas rutinas se apoya en un proceso residente que intercomunica las distintas máquinas virtuales: *pvmd* (*Parallel Virtual Machine Daemon*).

6.4.2.2 COHERENT VIRTUAL MACHINE

Coherent Virtual Machine (*CVM*) es una librería de programación en *C* que permite al usuario acceder a un sistema de memoria compartida, para de esta forma desarrollar aplicaciones distribuidas entre distintos procesadores [Keleher96]. La librería se ha desarrollado para sistemas *UNIX* y está enfocada principalmente para la intercomunicación de estaciones de trabajo. *CVM* está implementada en *C++*, y consiste en un conjunto de clases que establecen una interfaz básica e invariable de funcionamiento. Además, definen un protocolo de compartición de memoria flexible, un sistema sencillo de gestión de hilos y una comunicación entre aplicaciones distribuidas, desarrollado sobre *UDP* [Raman98].

Es posible implementar cualquier protocolo de sincronización de memoria compartida, derivando la implementación de las clases `Page` y `Protocol`. Todo funcionamiento que deseemos modificar respecto al comportamiento base, se especificará en los métodos derivados derogando su comportamiento [Booch94].

6.4.2.3 PERDIS

PerDis es un *middleware* que ofrece un entorno de programación de aplicaciones orientadas a objetos, distribuidas, persistentes y transaccionales, desarrolladas de una forma transparente, escalable y eficiente [Kloosterman98]. La creación de la plataforma *PerDiS* está motivada por la necesidad de desarrollar aplicaciones cooperativas de ingeniería en "empresas virtuales", entendiendo éstas como un conjunto de compañías o departamentos que trabajan conjuntamente en la implementación de un proyecto. Las distintas partes de la empresa virtual cooperarán y coordinarán su trabajo mediante la compartición de datos a través de su red de ordenadores.

PerDiS ofrece un entorno distribuido de persistencia, combinando características de modelos de memoria compartida, sistemas de archivos distribuidos y bases de datos orientadas a objetos. *PerDiS* permite, al igual que en los entornos de programación de memoria compartida [Li89], desarrollar aplicaciones que accedan a objetos en memoria indistintamente de su ubicación. Los objetos son enviados a la memoria local de un modo transparente cuando son requeridos, haciendo que la programación distribuida sea menos explícita en cuanto a la ubicación de los objetos.

Al igual que un sistema de archivos distribuido, *PerDiS* almacena en el disco bloques de datos denominados *clusters*, almacenando en el cliente una caché de los mismos. Un *cluster* es un grupo de objetos que define una unidad de almacenamiento, nombrado, protección y compartición, del mismo modo que sucede en los sistemas de archivos [Sun89]. *PerDiS* facilita además un conjunto básico de funcionalidades típicas de bases de datos orientadas a objetos [Bancilhon92] como el almacenamiento nativo y manipulación de objetos desde lenguajes orientados a objetos como *C++*, diferentes modelos de transacciones y tolerancia a fallos.

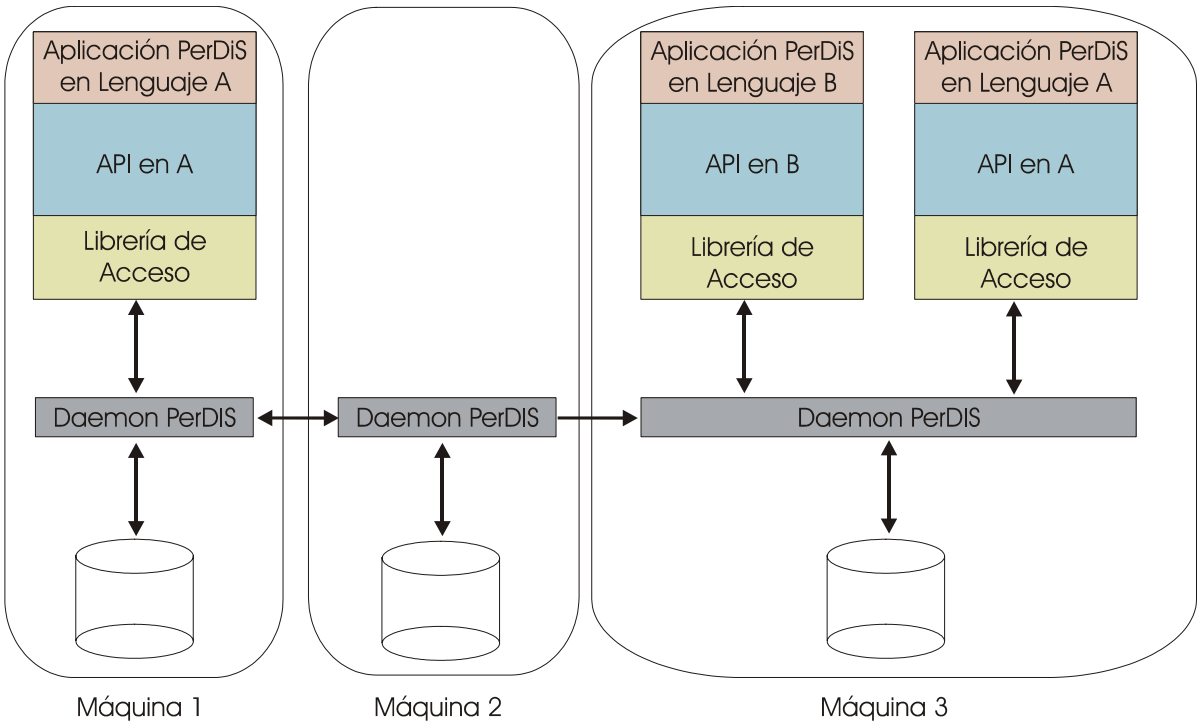

Figura 6.12: Arquitectura de la plataforma *PerDiS*

La plataforma *PerDiS* se basa en una arquitectura cliente/servidor simétrica: cada nodo (ordenador del sistema) se comportará al mismo tiempo como cliente y servidor. La Figura 6.12 muestra los componentes propios de la arquitectura del sistema:

- Una aplicación desarrollada sobre la plataforma virtual, en un lenguaje de programación.
- *API* (*Application Programming Interface*) de acceso a la plataforma, dependiente del lenguaje de programación.
- Una única librería, independiente del lenguaje, que sirve como nexo entre la plataforma virtual y el *API* utilizado para una determinada aplicación.
- El proceso demonio, "*PerDiS Daemon*", encargado de interconectar todos los ordenadores de la plataforma de forma independiente al lenguaje de programación, y gestor del sistema de persistencia distribuido.

El sistema es independiente del lenguaje, y en cada nodo existe parte del almacenamiento global del sistema, aunque no exista ninguna aplicación en ejecución.

6.4.2.4 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Los sistemas estudiados son ejemplos de sistemas que tratan de ofrecer una plataforma de computación virtual. Para ello, no definen una máquina abstracta única, sino que establecen una interfaz de acceso a cada máquina concreta. La unión de todos los accesos desde las distintas plataformas existentes consigue la abstracción de una única plataforma virtual, susceptible de ser utilizada como una sola entidad para la

resolución de un problema. El principal inconveniente de los sistemas estudiados reside en que su diseño ha estado enfocado a la resolución de un determinado problema y carece de varias características enunciadas en los requisitos para la máquina virtual que hemos establecido anteriormente en esta tesis.

6.4.3 Plataformas Independientes

En esta sección se analizará el uso de máquinas abstractas para conseguir plataformas que sean independientes tanto del lenguaje como del sistema operativo sobre el que corren y también del procesador. Estos sistemas se basan en una máquina abstracta sobre la que hacen toda su computación y se desarrolla toda su plataforma, abstrayéndose del entorno de computación físico real sobre el que se ejecutan.

6.4.3.1 SMALLTALK-80

El sistema *Smalltalk-80* se creó en la década de los setenta, implementándose tres sistemas principales: *Smalltalk 72, 76 y 80*, según el año en el que fueron diseñados [Krasner83]. Aunque ya hemos hablado de este sistema en un capítulo anterior, contemplaremos ahora aspectos relativos a su máquina virtual exclusivamente.

El principal objetivo de diseño en *Smalltalk* era obtener un sistema que fuese manejable por no informáticos. Para llegar a esto, se apostó por un sistema basado en gráficos, interfaces interactivas y visuales, y una mejora en la abstracción y flexibilidad a la hora de programar. La abstracción utilizada, más cercana al ser humano, era la orientación a objetos, y en lo referente a flexibilidad se podía acceder cómodamente y en tiempo de ejecución a toda clase y objeto que existiese en el sistema [Mevel87].

El sistema *Smalltalk-80* está dividido en dos grandes componentes [Goldberg83]:

- **La imagen virtual**: Colección de objetos e instancias de clases que proporcionan estructuras básicas de datos y control, primitivas de texto y gráficos, compiladores y manejo básico de la interfaz de usuario, entre otras funciones.
- **La máquina virtual**: Intérprete de la imagen virtual y de cualquier aplicación del usuario. A su vez está dividida en:
    - El gestor de memoria: Se encarga de gestionar los distintos objetos en memoria, sus relaciones y su ciclo de vida. Para ello implementa un recolector de basura para los objetos.
    - El intérprete de instrucciones: Analiza y ejecuta las instrucciones en tiempo de ejecución. Las operaciones que se utilizan son un conjunto de primitivas que operan directamente sobre el sistema.

Smalltalk no es un lenguaje interpretado directamente, ya que pasa primero por una fase de compilación a un código binario, en la que se detectan una serie de errores. Este código binario será posteriormente interpretado por el simulador o procesador *software* de la máquina abstracta. La especificación formal de la máquina abstracta se diseñó pensando precisamente en esta característica, lo que le permite una flexibilidad del sistema y un nivel de abstracción base adecuado.

Todas las aplicaciones del sistema *Smalltalk-80* están escritas en el propio lenguaje *Smalltalk,* y gracias a que éste tiene carácter interpretado se puede acceder dinámicamente a todos los objetos existentes en tiempo de ejecución. El mejor ejemplo es el *Browser* del sistema [Mevel87] mostrado en la Figura 6.13, donde se presenta una aplicación que recorre todas las clases (los objetos derivados de `Class`) del sistema (del diccionario `Smalltalk`) y visualiza la información de ésta, que incluye:

- Categoría a la que pertenece la clase.
- Un texto que describe la funcionalidad de ésta.
- Sus métodos.
- Sus atributos.
- Código que define el comportamiento de cada método.

Además, esta información es modificable en todo momento (reflexión estructural) y dispondremos también de documentación real, puesto que se genera dinámicamente, de toda clase, objeto, método y atributo existente en el sistema.

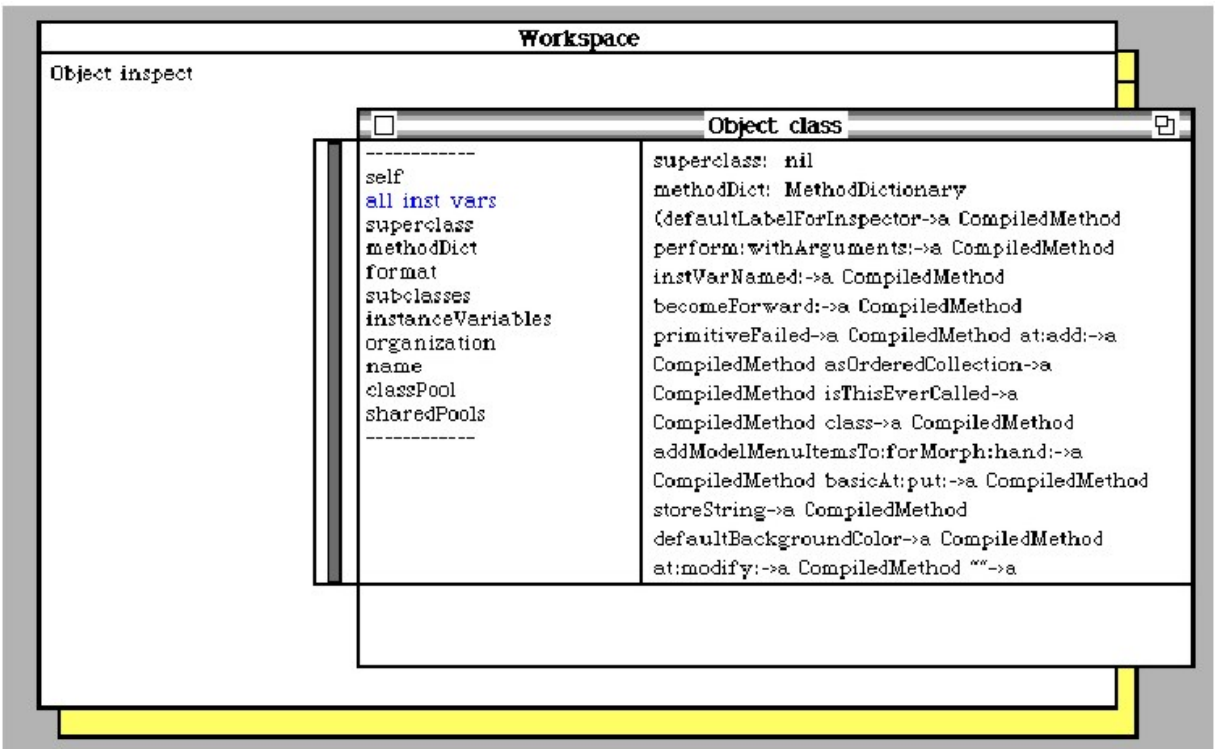

Figura 6.13: Acceso a un método `inspect` del objeto `Object` en *Smalltalk*

El aplicar estos criterios con todos los programas del sistema hace que la programación, depuración y análisis de éstos sea muy sencilla. Esto sólo es posible gracias a la presencia de una máquina abstracta, encargada de ejecutar el sistema. *Smalltalk-80* es pues un sistema de computación que consigue mediante una máquina virtual una integración entre todas sus aplicaciones, una independencia de la plataforma y un nivel de abstracción orientado a objetos para su modelo de computación base.

6.4.3.2 SELF

El proyecto *Self* [Ungar87] se creó a partir de la plataforma y lenguaje de *Smalltalk-80* originales. Se especificó una máquina abstracta que reducía la representación en memoria del modelo de objetos utilizado por *Smalltalk* y que se describió anteriormente. El criterio principal de diseño fue usar el modelo de objetos basado en prototipos [Evins94] que veremos en un capítulo posterior.

En la búsqueda de optimización de máquinas virtuales, se descubrieron nuevos métodos de compilación e interpretación, como la compilación continua unida a la optimización adaptable [Hölze94]. Estas técnicas de compilación y optimización avanzadas han sido utilizadas en la mejora de máquinas abstractas comerciales como la de *Java* (*The Java "HotSpot" Virtual Machine*) [Sun98].

El modo de programar en *Self* varía respecto a los entornos clásicos, ya que se basa en un sistema gráfico que permite crear aplicaciones de forma continua e interactiva. Posee un conjunto de objetos gráficos, denominados *morphs,* que facilitan la creación de interfaces, inspeccionándolos, modificándolos y volcándolos a disco. La programación se lleva a cabo accediendo, en tiempo de ejecución, a cualquier objeto existente en memoria, posea éste o no representación gráfica. Esta facilidad es definida como *"live editing"*. Además, los objetos existentes en el sistema son compartidos por todos los usuarios y aplicaciones.

Self se implementó únicamente en plataformas *Solaris*, y su utilización práctica se limitó a la investigación y análisis para la creación de otros sistemas más comerciales, como *Java*. Uno de los proyectos para los que se utilizó fue el proyecto *Merlin* que también aparece reflejado en esta tesis: Un sistema que trataba de acercar los ordenadores a la forma de pensar de los humanos, pudiendo ser éstos fácilmente utilizables por cualquier persona [Assumpcao93]. *Self* fue ampliado para obtener un grado de flexibilidad mayor, siendo capaz de modificar parte de su comportamiento mediante reflexión computacional [Assumpcao95]. La plataforma *Self* modificada pasó de ser un sistema con un modelo muy básico a ganar en flexibilidad y complejidad utilizando conceptos como "*mappings*" y "*reflectors*" que no veremos en esta tesis.

6.4.3.3 JAVA

En 1991, un grupo de ingenieros trabajaba en el proyecto *Green*: un sistema para interconectar cualquier aparato electrónico. Se buscaba poder programar cualquiera de ellos mediante un lenguaje sencillo, con un intérprete reducido y cuyo código fuese totalmente portable. Para ello especificó una máquina abstracta con un código binario en *bytes* y se trato de usar *C++* [Stroustrup98] como lenguaje de programación. No obstante este lenguaje resulto demasiado complejo y dependiente de la plataforma de ejecución, siendo reducido al lenguaje *Oak*, que luego evolucionaría a *Java* [Gosling96].

No obstante ninguna empresa se interesó en el producto, y en 1994 el proyecto había fracasado. En 1995, con el extensivo uso de *Internet*, desarrollaron en *Java* un navegador *HTTP* [Beners96] capaz de ejecutar aplicaciones *Java* en el cliente descargadas previamente del servidor, denominándose éste *HotJava* [Kramer96]. A raíz de esta implementación, *Netscape* introdujo en su *Navigator* el emulador de la máquina abstracta permitiendo añadir computación, mediante *applets*, a las páginas estáticas *HTML* utilizadas en *Internet* [Beners93]. A partir de entonces, la tecnología *Java* comenzó su constante expansión hasta resultar mundialmente conocida y ampliamente usada.

Un programa en el lenguaje *Java* se compila para ser ejecutado sobre una plataforma independiente [Kramer96]. Esta plataforma de ejecución independiente está formada básicamente por:

- La máquina virtual de *Java* (*Java Virtual Machine*) [Sun95].
- La interfaz de programación de aplicaciones en *Java* o *Core API* (*Application Programming Interface*).

Esta dualidad, que consigue independencia de la plataforma y un mayor nivel de abstracción en la programación de aplicaciones, es idéntica que la presentada en *Smalltalk-80*: imagen y máquina virtual. El *API* es un conjunto de clases que están compiladas en el formato de código binario de la máquina abstracta [Sun95]. El programador puede usar estas clases para crear aplicaciones de una forma más sencilla.

La máquina virtual de *Java* es el procesador del código binario de esta plataforma. El soporte a la orientación a objetos está definido en su código binario aunque no se define en su arquitectura. Por lo tanto, la implementación del intérprete y la representación de los objetos en memoria quedan a disposición del diseñador del simulador, que podrá utilizar las ventajas propias de la plataforma real.

La creación de la plataforma de *Java* se debió a la necesidad existente de abrir el espacio computacional de una aplicación. Las redes de computadores interconectan distintos tipos de ordenadores y dispositivos entre sí y, aprovechando las posibilidades que ofrecen, se han creado múltiples aplicaciones, protocolos, *middlewares* y arquitecturas cliente-servidor para resolver el problema de la programación distribuida [Orfali96]. Una red de ordenadores tiene interconectados distintos tipos de dispositivos y ordenadores, con distintas arquitecturas *hardware* y sistemas operativos. *Java* define una máquina abstracta para conseguir implementar aplicaciones distribuidas que se ejecuten en todas las plataformas existentes en la red, logrando por tanto independencia de la plataforma. De esta forma, cualquier elemento conectado a la red, que posea un procesador de la máquina virtual y el *API* correspondiente, será capaz de procesar una aplicación (o una parte de ésta) implementada en *Java*. Para conseguir este tipo de programación distribuida, la especificación de la máquina abstracta de *Java* se ha realizado siguiendo fundamentalmente tres criterios [Venners98]:

- Búsqueda de una plataforma independiente.
- Movilidad del código a lo largo de la red.
- Especificación de un mecanismo de seguridad robusto.

La característica de definir una plataforma independiente está implícita en la especificación de la propia máquina abstracta. Sin embargo, para conseguir la movilidad del código a través de las redes de computadores, fue necesario tener en cuenta otra cuestión: la especificación de la máquina ha de permitir obtener código de una aplicación a partir de una máquina remota. La distribución del código de una aplicación *Java* es directamente soportada por la funcionalidad de los "*class loaders*" [Gosling96]. Mediante el empleo de estos elementos se permite definir la forma en la que se obtiene el *software* (en este caso las clases), permitiendo que éste pueda estar localizado en máquinas remotas. De esta forma, la distribución de *software* es automática, puesto que se puede centralizar todo el código en una sola máquina y ser cargado desde los propios clientes al principio de cada ejecución.

Otras características de *Java* también muy importantes es su ya mencionado soporte para introspección (mediante el paquete `java.lang.reflect`), permitiendo la consulta de los atributos y métodos de cualquier objeto y clase que se defina en el sistema, de forma sencilla y completamente integrada en el lenguaje. Además, se permite la carga dinámica de clases (y por tanto de código no existente en tiempo de compilación) mediante el empleo de los *ClassLoaders* mencionados anteriormente, para

de esta forma lograr una mayor flexibilidad que, aunque siempre bajo una estricta política de seguridad configurable por el usuario que evite la ejecución accidental de código malicioso.

6.4.3.4 LA MÁQUINA VIRTUAL *J9* DE *IBM*

Como una implementación comercial de la máquina abstracta de *Java*, la máquina virtual *J9* de *IBM* está enfocada al desarrollo de *software* empotrado [IBM00b]. Mediante la utilización de esta máquina abstracta y el entorno de desarrollo "*VisualAge Micro Edition*", se puede implementar *software* estándar que sea independiente de las siguientes variables [IBM00c]:

- El dispositivo *hardware* destino (microprocesador).
- El sistema operativo destino.
- Las herramientas utilizadas en el desarrollo de la aplicación.

"*VisualAge Micro Edition*" ofrece un amplio abanico de librerías de clases para ser implantadas: desde librerías de tamaño adecuado a dispositivos de capacidades reducidas, hasta amplias *APIs* para los entornos más potentes. Con esta arquitectura, el código de la aplicación está aislado de las peculiaridades existentes en la plataforma física destino. Mediante el "*Java Native Interface*" *(JNI)* [Sun97c], se podrá acceder directamente a los controladores de los dispositivos y a funcionalidades propias de sistemas operativos en tiempo real, aislándose el código nativo del portable.

Este sistema de *IBM* es pues un ejemplo práctico de la portabilidad de código para la plataforma *Java*. Define un sistema estándar para la creación, ejecución y embebido de aplicaciones empotradas en *hardware* [IBM00d].

6.4.3.5 *JALAPEÑO* Y *RVM*

Jalapeño [IBMResearch05] es una máquina virtual de *Java* creada por *IBM Research* e inicialmente optimizada para servidores, es decir, es una máquina virtual de *Java* diseñada desde cero teniendo en cuenta las especiales características de ordenadores que actúan como servidor, diferenciando su construcción, requisitos y arquitectura de las máquinas virtuales similares destinadas eminentemente para clientes o uso personal. Las características tenidas en cuenta para su diseño son:

- Se tiene en cuenta que las disponibilidades de memoria principal de un servidor son mayores que las de un cliente.
- Se optimiza el compilador *JIT* de la máquina de manera que explote las características de procesadores de alta capacidad, optimizando las aplicaciones para sacar partido de la arquitectura de memoria, paralelismo de instrucciones y otras características avanzadas.
- Se proporciona soporte para procesadores múltiples que comparten acceso a memoria principal (*SMP*), optimizando la creación de hilos (por ejemplo) y no estableciendo límites muy restrictivos para los mismos.
- Mejora del tiempo de respuesta a las peticiones.

- Mejora de la capacidad de proporcionar servicios durante largos periodos de tiempo sin una degradación muy elevada, algo muy importante en servidores.

Para cumplir con estos requisitos, esta plataforma incorpora un conjunto de características en su diseño, entre las que destaca un compilador que ejecuta optimizaciones de forma dinámica, ayudando a eliminar cuellos de botella en la ejecución. Este compilador permite aplicar optimizaciones avanzadas sólo en aquellas partes que se detecten como cuellos de botella en la implementación. También se ofrece como una plataforma para probar nuevas ideas en la construcción de máquinas virtuales.

Debido a que su diseño está vinculado a servidores y que la construcción de la máquina se ha hecho aprovechándose de ciertas características arquitectónicas, esta máquina no estuvo diseñada en principio pensando en su portabilidad hacia otras plataformas. La mayor parte está escrita en *Java*, con una pequeña parte en *C*.

Este sistema es pues una implementación de una *JVM* optimizada para servidores, con una serie de características que optimizan su rendimiento y características frente a otras implementaciones destinadas a clientes, y que además permite servir como base para pruebas y *test* de nuevos avances en el campo de las máquinas virtuales. Posteriormente, el código correspondiente a este proyecto se liberó, creándose un proyecto derivado de esta máquina llamado *Jikes RVM* [JikesRVM05], creada con el propósito de servir como base para la el desarrollo e implementación de nuevos diseños que sirvan para la construcción y experimentación con nuevos avances en el campo de las máquinas virtuales. Ésta es una máquina licenciada bajo licencia *CPL*, y aprobado como un producto con licencia de código abierto.

Esta última máquina si está disponible para diferentes arquitecturas, y al igual que su proyecto "madre" proporciona tecnologías de compilación dinámica, optimización adaptable, recolección de basura y sincronización y planificación de hilos avanzada. Es también una máquina implementada en *Java* que permite la ejecución del código *Java*, sin necesidad de contar con una segunda máquina virtual, y que goza de popularidad como plataforma de pruebas para nuevos avances en el cambio de máquinas virtuales, que, como ya se ha dicho, era uno de los propósitos principales de su creación.

6.4.3.6 .NET

ARQUITECTURA Y DISEÑO GENERAL

.NET [Microsoft05] es la estrategia de *Microsoft* para la interconexión de información, personas, sistemas y dispositivos mediante *software*, basándose en herramientas como los servicios *web*. Su objetivo es proporcionar un marco de desarrollo que permite construir, desplegar, mantener y usar rápidamente soluciones que tengan opciones de seguridad mejoradas.

En la base de la arquitectura de la plataforma *.NET* se encuentra una máquina virtual denominada *"Common Language Runtime" (CLR),* que proporciona un motor de ejecución de código independiente del sistema en el que fue desarrollado. Esta máquina es la implementación de *Microsoft* del estándar *Common Language Infraestructure* o *CLI* (*ECMA-335*). Sobre la raíz de la plataforma *(Common Runtime Language)*, se ofrece un marco de trabajo base: la librería base común (*BCL*), válida para cualquier plataforma, que puede ser utilizada desde cualquier lenguaje de programación, y que permite que cualquier lenguaje destinado a esta plataforma pueda tener acceso a las mismas funcionalidades. Por otra parte, el entorno de programación está basado en *XML*

[W3C98]. A éste se puede acceder desde una aplicación *Web* (con un navegador de *Internet*) o bien desde una interfaz gráfica. La comunicación entre cualquier aplicación se lleva a cabo mediante mensajes codificados en *XML*, siendo así independiente del programa y sistema operativo emisor [Skonnard01].

El diseño del *framework* se hizo con los siguientes objetivos [MSDNNET3006b]:

- **Interoperabilidad**: Todos los lenguajes que se ejecutan sobre la máquina pueden interoperar entre si, es decir que, además del uso común de la librería *BCL* ya mencionado, cualquier código realizado en uno de ellos puede ser empleado desde otro. Además, *.NET* ofrece la posibilidad de usar código heredado, implementado fuera de lo que es el propio entorno *.NET*, con características como *P/Invoke*.
- **Motor de ejecución común**: Todos los lenguajes de programación del *framework .NET* compilan a un lenguaje intermedio conocido como *CIL* (*Common Intermediate Language*). La implementación de *Microsoft* de este lenguaje intermedio se puede denominar también *MSIL* (*Microsoft Intermediate Language*). En *.NET* el lenguaje intermedio no es interpretado sino compilado con un compilador *Just-In-Time* (*JIT*). Este compilador *JIT* permite la optimización de aplicaciones en tiempo de ejecución, de manera que, mediante las operaciones que veremos en el capítulo siguiente dedicado a este tipo de técnicas, el rendimiento de las aplicaciones ejecutadas con el *framework* se verá incrementado significativamente. El compilador *JIT* y el lenguaje intermedio mencionados son piezas clave del motor de ejecución *CLR* del sistema.
- **Independencia del lenguaje**: El *framework .NET* introduce el concepto de *Common Type System* (*CTS*). Éste define todos los posibles tipos de datos soportados por el *CLR* y cómo pueden interactuar entre ellos. Esta característica es clave para permitir la interoperabilidad de lenguajes antes mencionada. Además, el framework *.NET* implementa el *Common Language Specification* (*CLS*) que describe una serie de características que todos los lenguajes que se ejecuten sobre el mismo deben tener en común. *CLS* incluye un subconjunto del *CTS,* y un programa que use tipos que cumplan con las normas del estándar *CLS* podrá interoperar con cualquier programa *.NET* escrito en uno de los lenguajes soportados por el mismo.
- ***BCL***: Como ya se ha mencionado anteriormente, ésta es una librería de tipos que está disponible para cualquier lenguaje del *framework*. *BCL* incorpora clases que encapsulan un conjunto de funciones típicas (interacción con bases de datos, manipulación de documentos *XML*, colecciones, ….), proporcionando un conjunto de características muy amplio, útiles para el desarrollo de aplicaciones. Está escrita principalmente en *C#* y puede ser usada por cualquier otro lenguaje diseñado e integrado en el sistema, de manera que cualquier lenguaje pueda acceder a las mismas funcionalidades. La librería de clases está estructurada en espacios de nombres (*Namespaces*) y se distribuye en librerías compartidas llamadas ensamblados (*Assemblies*). Un espacio de nombres es un mecanismo para la agrupación lógica de clases similares dentro de una estructura jerárquica, de manera que se puedan evitar conflictos de nombre, mientras que un ensamblado es un empaquetado físico de las librerías de clases. Los espacios de nombres se suelen distribuir dentro de varios ensamblados y es perfectamente posible que un ensamblado contenga múltiples archivos.
- **Seguridad**: El *framework* permite que el código pueda ser ejecutado con diferentes niveles de confianza y cuenta con características de seguridad más avanzadas.
- **Despliegue simplificado**: La instalación y despliegue de aplicaciones desarrolladas con el *framework* se ha simplificado mucho al eliminar la necesidad de modificar el registro del sistema y muchos problemas existentes anteriormente

con las librerías de enlace dinámico (*DLL*).

El diseño del *framework .NET* permite la independencia de la plataforma. Un programa escrito con el mismo debería funcionar sin cambios en cualquier computador donde exista una implementación de dicho *framework* (ejemplo de estas implementaciones es el sistema *Mono* que veremos a continuación). La siguiente figura muestra el funcionamiento general de *.NET*:

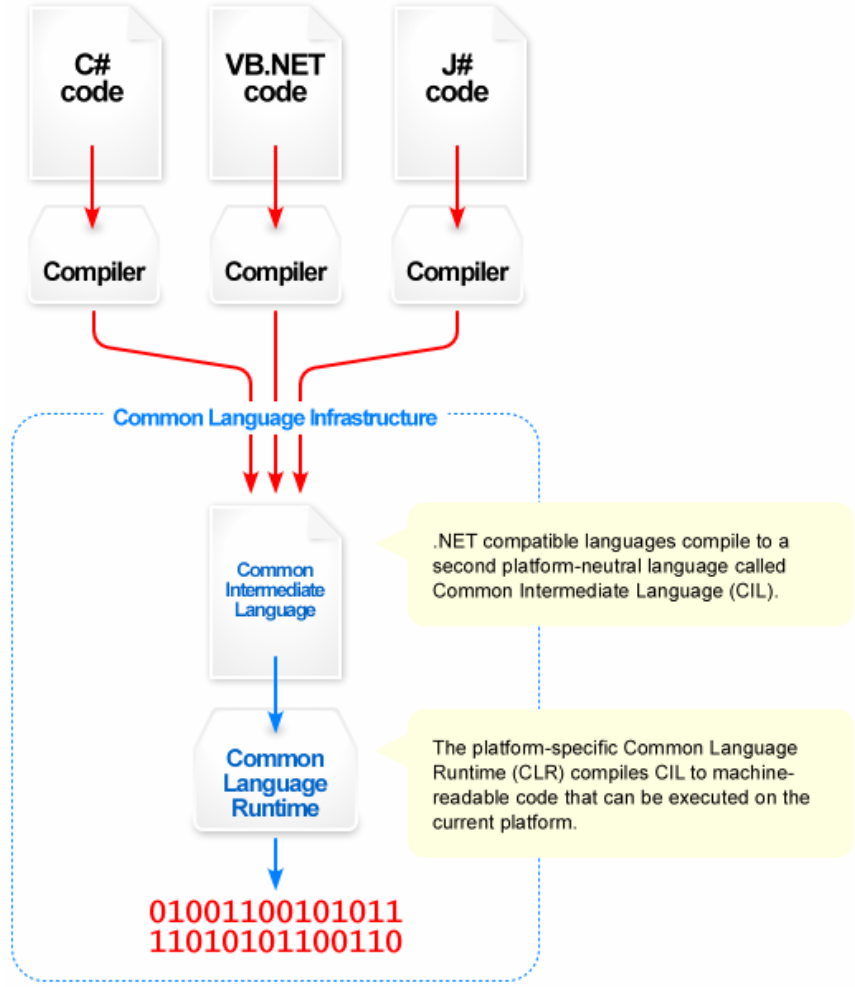

Figura 6.14: Esquema general del funcionamiento de .NET [WikiDNET06b]

Las ventajas propias de desarrollar aplicaciones en *.NET* son [Johnston00]:

- Utilización del mismo código base en *.NET*, puesto que su codificación es independiente de la máquina *hardware*.
- La depuración de aplicaciones se realiza de forma indiferente al lenguaje de implementación utilizado. Se pueden depurar aplicaciones, incluso si están codificadas en distintos lenguajes de programación.
- Los desarrolladores pueden reutilizar cualquier clase, mediante herencia o composición, indistintamente del lenguaje empleado.
- Unificación de tipos de datos y manejo de errores (excepciones).

- La máquina virtual ejecuta código con un sistema propio de seguridad.
- Se puede examinar cualquier código de una clase, obtener sus métodos, mirar si el código está firmado, e incluso conocer su árbol de jerarquía.

.NET FRAMEWORK 3.0

La versión del *.NET Framework* 3.0 [MSDNNET3006], también llamada *WinFX*, incluye un nuevo conjunto de *APIs* que son parte integral de la nueva versión de los sistemas operativos *Windows* (*Windows Vista*) y de *Windows Server* "*Longhorn*". No obstante, también está disponible para *Windows XP SP2* y *Windows Server 2003* como descarga separada. Esta versión no contiene características arquitectónicas nuevas e incluye la versión 2.0 del *CLR*. La disposición general de elementos en esta nueva versión puede verse en la siguiente figura:

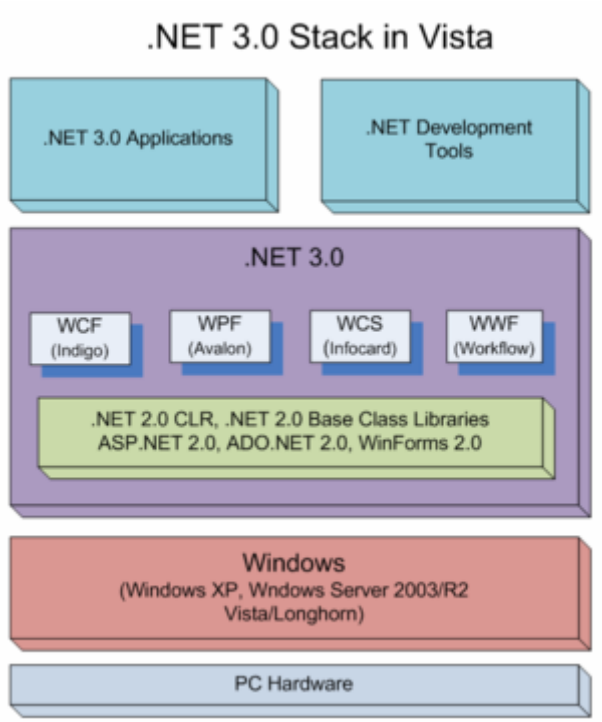

Figura 6.15: Arquitectura de .NET 3.0

Consta de cuatro componentes principales:

- ***Windows Presentation Foundation* (*WPF*) (*Avalon*)**: Una nueva interfaz de usuario y *API* basado en *XML* y gráficos vectoriales, que hará uso de las capacidades 3D de las tarjetas de video modernas, así como de tecnologías *Direct3D*.
- ***Windows Communication Foundation* (*WCF*) (*Indigo*)**: Un nuevo servicio de mensajería que permitirá a los programas interoperar de manera local o remota de forma similar a los servicios *web*.
- ***Windows Workflow Foundation* (*WWF*)**: Permite automatizar tareas y transacciones integradas mediante el uso de *workflows*.
- ***Windows CardSpace (WCS) (InfoCard)***: Un componente *software* que permite

guardar la identidad digital de una persona y permite conseguir una interfaz común para seleccionar la identidad de un sujeto que va a realizar una transacción concreta (como darse de alta en un sitio *web*).

CONCLUSIONES

Esta plataforma goza actualmente de una gran extensión a todos los niveles, y se puede afirmar que los propósitos por los cuales se creó se han alcanzado plenamente. Hoy en día la plataforma *.NET* ofrece gran cantidad de servicios (servicios *Web*, mensajería instantánea, etc.) que gozan de una amplia extensión y se ha convertido en una de las plataformas más extendidas en la red, siendo un competidor directo de la plataforma *Java*.

Como ya hemos visto anteriormente, esta plataforma cuenta además con soporte para características reflectivas dentro de la propia máquina, incluyendo capacidades de introspección y programación generativa que la dotan de una gran flexibilidad. Además, basándose igualmente en los documentos que especifican el estándares de diseño del sistema y de todos sus componentes (*ECMA-334* (*C#*) y *ECMA-335* (*CLI*)) [ECMA33405] [ECMA02] se han creado otras iniciativas, como *Shared Source Common Language Infraestructure* (*SSCLI,* llamada también *Rotor*) [MicrosoftSSCLI06], destinada principalmente a ser un vehículo de experimentación con dicho estándar y servir como una base para la implementación y prueba de nuevas tecnologías, investigaciones y avances sobre la misma.

6.4.3.7 MONO

Mono [Mono06] es un proyecto inicialmente impulsado por *Novell* [Novell06], y conocido inicialmente como *Ximian*, cuyo propósito es crear un conjunto de herramientas compatibles con los estándares *.NET* (que como ya se ha dicho es a su vez una implementación del estándar *CLI* [ECMA33405] [ECMA02]), siendo *Mono* una implementación del mismo estándar y sus tecnologías asociadas. *Mono* es a su vez portable y está disponible en un número mayor de plataformas como *Linux*, *FreeBSD*, *UNIX*, *MacOS X*, *Solaris* y *Windows*.

El código de *Mono* está disponible para desarrolladores de una forma más sencilla que el de *.NET*. El compilador de *C#* y otras herramientas tienen una licencia *GPL* (*GNU General Public License*) [GNU06], mientras que las librerías en tiempo de ejecución se encuentran licenciadas mediante la licencia *LGPL* (*GNU Lesser General Public License*) [GNU06b] y las librerías de clases se encuentran dentro de la licencia *MIT* [OpenSource06]. Todas estas licencias convierten a *Mono* en un proyecto esencialmente de código abierto y son diferentes a la licencia *Shared Source* que licencia *SSCLI* [MicrosoftSSCLI06], que por ejemplo impide su uso comercial.

Además de esto, la máquina virtual de *Mono* incluye motores de compilación *JIT* (*Just In Time*) para diferentes procesadores (*SPARC*, *ARM*, *PowerPC*, *ARM* y *x86* en 32 *bits* y también para algunas plataformas de 64 *bits*). Mientras la plataforma es capaz de convertir el código *CIL* a código nativo para estas plataformas usando el compilador integrado, para otras plataformas diferentes se requerirá el uso de intérpretes.

El principal promotor de esta plataforma es Miguel de Icaza, iniciando su investigación en la plataforma a finales del año 2000 y comenzando el desarrollo de la plataforma *Mono*, cuya versión inicial (1.0) está disponible desde Junio de 2004, casi tres años después. La arquitectura simplificada de *Mono* se muestra en la figura siguiente, donde se pueden ver sus principales componentes.

Al igual que en el caso de la plataforma de *Microsoft*, en *Mono* el *Common Language Infraestructure* o *CLI* es usado para ejecutar las aplicaciones *.NET* que han sido compiladas y siguen las normas definidas por el estándar *ECMA-335*. El compilador de *Mono* genera una imagen que sigue la especificación del *CLS* ya mencionada anteriormente. Esta imagen es la que finalmente procesa el entorno de ejecución del sistema.

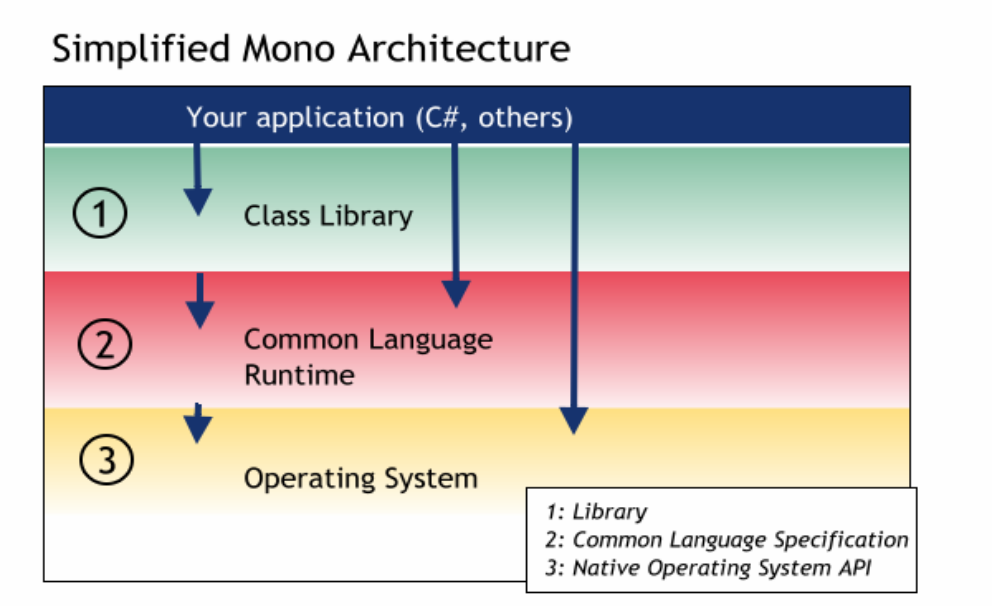

Figura 6.16: Diagrama simplificado de la arquitectura de *Mono* [WikiMono06]

APLICACIONES

En lo referente a las aplicaciones que se pueden ejecutar dentro de esta plataforma, en una aplicación *Mono* (al igual que en *.NET*) todo el código se dice que es *managed*, entendiendo por ello que es el *CLI* el que controla el uso de la memoria que la aplicación realiza y la seguridad de los hilos de ejecución. No obstante cualquiera de estas aplicaciones puede usar código ya existente fuera de la plataforma *.NET*, código que se denomina *unmanaged*, mediante las herramientas proporcionadas para ello.

Aunque puede decirse que *Mono* está menos extendido que la plataforma *.NET* (no hay que olvidar que esta última plataforma soporta una gran cantidad de servicios y de funcionalidades que están integrados en los sistemas operativos de la familia *Windows*, y su área de distribución es potencialmente mucho más grande), también se usa en proyectos de importancia, como *Cocoa#* [Cocoa06], usado entre otras cosas para permitir usar el núcleo del sistema *Mono* en *Mac OS X* o *Gtk#* [Gtk05], que permite el uso de las librerías *Gtk+*, implementadas en *C*, todo ello teniendo en cuenta que la popularidad y ámbito de aplicación de la plataforma se espera que aumente según vaya evolucionando la plataforma y las funcionalidades aportadas por las sucesivas versiones.

Mono es pues una implementación alternativa a la proporcionada por *Microsoft* y que puede proporcionar una funcionalidad equivalente, aunque en la actualidad este proyecto no soporta todas las características del primero. No es la única posibilidad existente, ya que en este sentido existen otros proyectos como *DotGNU* [DotGNU05], que también ofrece una implementación alternativa de similares características y que será descrita más en detalle en un capítulo posterior.

6.4.3.8 PARROT

Parrot [Parrot06] es una máquina virtual basada en registros que está siendo actualmente desarrollada desde cero con la idea de soportar inicialmente el lenguaje *Perl 6*. La intención futura es que esta máquina pueda ser empleada idealmente para ejecutar *bytecode* de múltiples lenguajes (no sólo *Perl*), como *Python*, *Ruby* o *Tcl*, lo que implica que en su diseño se deba contemplar un alto grado de flexibilidad para satisfacer las demandas de todos estos lenguajes.

El desarrollo de esta máquina se realiza en lenguaje *C*, y está optimizada para la ejecución eficiente de lenguajes dinámicos (en contraposición con *JVM* o *CLI*, diseñadas para la ejecución eficiente de lenguajes estáticos). Ésta es la razón por la cual se ha desarrollado una máquina virtual propia en lugar de usar una existente, ya que la mayoría de máquinas disponibles están optimizadas para lenguajes estáticos y ejecutar un lenguaje dinámico sobre una de estas máquinas tendría un impacto adverso en su rendimiento.

La máquina emplea la técnica de compilación *JIT* para mejorar el rendimiento global de las aplicaciones. Actualmente es posible compilar solamente algunos lenguajes (como *PASM* (*Parrot Assembly Language*) y *PIR* (*Parrot Intermediate Language*), que mencionaremos posteriormente) a *bytecode* propio de *Parrot* y ejecutarlos. El subsistema de compilación *JIT* de *Parrot* [CpanParrot06] convierte el *bytecode* de la máquina en instrucciones nativas del procesador justo antes de que éstas se ejecuten, eliminando el coste adicional introducido por la interpretación del código. Cada procesador requiere su propio compilador *JIT*. Actualmente existen *JIT* para las plataformas *DEC Alpha*, *ARM*, *Intel i386*, *SGI MIPS*, *PPC* y *Sun4*.

Esta máquina también cuenta con un sistema de ejecución nativa, que permite la integración del entorno de ejecución de *Parrot* y de un programa *Parrot* en un solo binario precompilado, reduciendo el tiempo de arranque del programa y permitiendo que se ejecute sin instalar la máquina *Parrot* independientemente. Esta característica también hace uso del subsistema *JIT* mencionado y sólo está soportada hasta el momento en plataformas *x86*.

El hecho de que esta máquina esté basada en registros (y no en una pila, como *JVM*) no es casual, sino que su intención es que, al tener una arquitectura más próxima al *hardware* real existente, se puedan emplear en ella todas las técnicas de optimización de compiladores destinadas a este tipo de arquitecturas, para generar código lo más eficiente posible e intentar aproximar su rendimiento al ofrecido por los lenguajes estáticos. Esta máquina está cubierta por una licencia *GPL* [GNU06] de *GNU* y goza de un extenso soporte para diferentes sistemas operativos (*Unix*, *Windows*, *Mac OS X*, etc.) y *CPUs* (*x86*, *SPARC*, *ARM*, etc.). En la actualidad, dentro del sistema *Parrot* existen tres tipos de código:

- ***Bytecode***: Interpretado nativamente por *Parrot*.
- ***PASM*** (*Parrot Assembly Language*): Lenguaje de bajo nivel de la máquina que se compila a *bytecode*.
- ***PIR*** (*Parrot Intermediate Representation*): Un lenguaje de un nivel superior a *PASM* y que será aquél al que los lenguajes implementados sobre esta máquina deberán compilarse.

Este proyecto es por tanto una implementación, aún en construcción, muy interesante de una máquina virtual destinada a soportar todo un elenco de lenguajes dinámicos en el futuro y, de cumplirse todos los objetivos planteados en su desarrollo, se contará con un sistema potente que permitirá la ejecución bajo una misma plataforma de

una serie de lenguajes de forma similar a lo que se plantea en el *CLI*, aunque la posibilidad de que también pueda ofrecer interoperabilidad entre todos los lenguajes no ha podido ser constatada en la actualidad, ya que el proyecto aún está en sus primeras fases de construcción.

6.4.3.9 ZERO

La máquina virtual *Zero [Schofield06]* está inspirada en "*The Design and Evolution of C++*", de Bjarne Stroustrup [Stroustrup94]. Esta máquina fue creada con el propósito de ser un medio de aprendizaje de la programación orientada a objetos basada en prototipos, demostrando también qué beneficios podrían obtenerse mediante el uso de la persistencia. En concreto, se pueden definir los objetivos de esta máquina virtual en los siguientes puntos:

- Proporcionar a los estudiantes un sistema orientado a objetos puro, en el que sea posible incluir el soporte "ortogonal" de persistencia basado en contenedores. Será posible, además, crear diferentes compiladores para este sistema.
- Proporcionar un lenguaje orientado a objetos puro, basado en prototipos, que sea de utilidad para la docencia. La máquina no soporta tipos a bajo nivel.
- Para que los anteriores objetivos sean sencillos de cumplir, *Zero* nace con la idea de ser un sistema minimalista: es decir, pequeño y sencillo, por encima de cualquier otra consideración, incluyendo el rendimiento.

El sistema Zero, se compone de:

- Macroensamblador: Este sistema ha implementado un macroensamblador para permitir abordar la creación de programas de un nivel de complejidad superior, al ser un lenguaje que proporciona un nivel de abstracción superior al del ensamblador de la máquina puro. No obstante, este macroensamblador no es un lenguaje de programación, luego no se puede usar para proyectos a gran escala.
- Ensamblador: Permite las operaciones básicas directamente sobre la máquina virtual, como la creación de objetos, envío de mensajes y control de excepciones.
- Máquina virtual: Ejecuta los objetos creados por el ensamblador, el macroensamblador, o los lenguajes *Prowl* o *J--*. La máquina virtual es multiplataforma, existiendo versiones para Linux y Windows.
- Librería estándar interna: La librería estándar de *Zero*, que contiene una serie de tipos y objetos para facilitar el desarrollo de programas sobre esta plataforma.

Las características básicas de esta máquina virtual son: herencia simple, dinámica (implementada mediante delegación), creación y clonación de objetos (y prototipos, indistinguibles de los primeros), paso de mensajes, manejo de excepciones y persistencia basada en contenedores.

La máquina virtual está basada en registros (que guardan referencias a objetos), estructurándose en dos grandes grupos: el acumulador (`__acc`), que guarda la referencia resultado de la instrucción anterior, el registro que guarda el objeto que está ejecutando el método (`__this`), y el registro que guarda la excepción que se haya producido (`__exc`). En un segundo grupo están los registros generales que pueden ser utilizados para cualquier propósito (`__gpn`). Para facilitar el desarrollo de aplicaciones se han

desarrollado dos compiladores de lenguajes de alto nivel: uno, *J--*, es un subconjunto de *Java*, mientras el otro es similar a *Self* [Chambers89].

Lo más interesante de esta plataforma es el modelo de objetos que utiliza, basado en prototipos y dotado de reflectividad estructural en tiempo de ejecución, y cómo aprovecha éste para implementar un mecanismo de persistencia basado en contenedores con evolución del esquema.

En el modelo computacional basado en prototipos que utiliza *Zero* no existe el concepto de clase; la única abstracción existente es el objeto [Borning86]. Un objeto describe su estructura (conjunto de atributos), su estado (los valores de éstos) y su comportamiento (la implementación de los métodos que pueda interpretar).

La herencia es un mecanismo jerárquico de delegación de mensajes existente también en el modelo de prototipos ya visto: Si se le envía un mensaje a un objeto, se analiza si éste posee un método que lo implemente y, si así fuere, lo ejecuta; en caso contrario se repite este proceso para sus objetos padre, en el cado de que los hubiere. La relación de herencia entre objetos basados en prototipos es una asociación más, dotada de una semántica adicional –la especificada en el párrafo anterior. En el caso de *Zero*, se sigue el criterio del lenguaje de programación *Self* [Ungar87]; la identificación de esta semántica especial es denotada por la definición del miembro `parent`. El objeto asociado mediante este miembro es realmente el objeto padre. Al tratarse la herencia como una asociación, es posible modificar en tiempo de ejecución el objeto padre al que se hace referencia, obteniendo así un mecanismo de herencia dinámica. También existe soporte para la creación de objetos *trait* ya explicados en el capítulo dedicado a los modelos de objetos basados en prototipos.

Esta máquina virtual ofrece por tanto soporte para los conceptos de objeto rasgo y objeto prototipo ya vistos, y por tanto la creación de nuevos objetos se hace mediante clonación del prototipo. El mecanismo empleado para clonar los objetos es la reflectividad estructural. Además de las ventajas que aporta la reflectividad a un modelo computacional, ésta permite implementar un mecanismo de evolución del esquema no soportado por el modelo de clases de un modo coherente [Schofield2002].

Esta máquina tiene un soporte integrado para la persistencia a partir de un sistema de contenedores que utiliza, basado en la composición jerárquica de objetos. El objeto raíz del sistema de persistencia es denominado `psRoot`. El objeto *trait* que es utilizado para crear nuevos contenedores (clones de un hijo prototipo) es el contenedor `psRoot` que es, de hecho, un objeto derivado de este *trait*. Mediante la adición de atributos a los objetos contenedores, mediante reflectividad estructural, se hacen persistentes los atributos añadidos (persistencia por alcance). Si a un contenedor se le añade otro contenedor, se establecerá una composición jerárquica de contenedores. El siguiente ejemplo muestra las capacidades de persistencia de esta máquina:

```
object DisneyPersistente

method + doIt()
{
        reference disney = Container.createChild( "Disney" );
        reference donald = Persona.createChild( "donald" );
        reference daisy = Persona.createChild( "daisy" );

        donald.ponNombre( "Donald" );
        daisy.ponNombre( "Daisy" );
        disney.add( donald );
        disney.add( daisy );
        psRoot.add( disney ;

        return;
}
endObject
// * Los objetos donald y daisy ya son persistentes
```

Esto hará que el contenedor *disney*, que cuelga del contenedor raíz del almacenamiento persistente (*psRoot*) se guarde con los objetos que contenga. Una vez éste proceso termina, y el contenedor está por tanto guardado en el almacenamiento persistente, siendo posible recuperarlo mediante una referencia simple en cualquier otro proceso (*psRoot.Disney.donald*). Esto se aprecia en el siguiente código:

```
object MostrarDisney

method + doIt()
{
        System.console.write( psRoot.Disney.donald.toString() );
        System.console.lf();
        return;
}
endObject
```

La máquina virtual de *Zero* posee una interfaz gráfica llamada *Visual Zero VM* [Schofield06b], en la que se puede manipular dicha máquina y los objetos que contiene, permitiendo añadirles atributos y métodos (reflexión estructural), crear nuevos objetos a voluntad y otras operaciones, de una forma similar a como se ha visto con el lenguaje *Smalltalk*, todo ello facilitado por el empleo del modelo de objetos orientado a prototipos. Mediante un sistema como éste, se logra un soporte completo de persistencia de una forma transparente y completamente integrada en el sistema, beneficiándose dicho sistema de esta característica sin necesidad de incluir construcciones explicitas para su soporte en los lenguajes diseñados para esta plataforma. Además, el uso del modelo de prototipos hace que la implementación de características de reflexión estructural no sea demasiado compleja, ya que este modelo se adapta mejor a operaciones de esta clase que el modelo de clases tradicional. No obstante, este sistema está pensado para ser una herramienta de aprendizaje no comercial, muy simple y su rendimiento no ha sido un objetivo primario a la hora de su diseño.

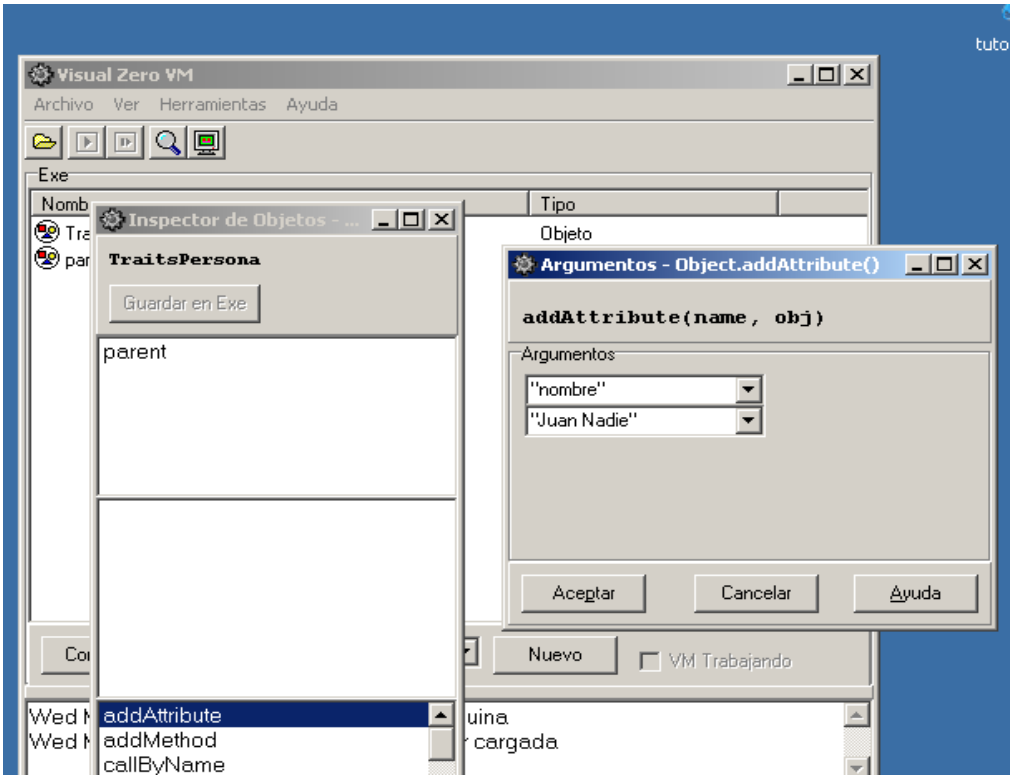

Figura 6.17. Entorno de Programación *Visual Zero* y su modelo de prototipos

6.4.3.10 SCUMMVM

ScummVM (*Script Creation Utility for Maniac Mansion Virtual Machine*) [SCUMMVM06] es una máquina virtual de pila multiplataforma y desarrollada casi en su totalidad en *C++*, que permite ejecutar aventuras gráficas que usan el motor *SCUMM*, (creadas por la compañía *LucasArts*), además de una serie de juegos adicionales de otras compañías y que no comparten la tecnología empleada por la primera. Es un *software* gratuito bajo licencia *GNU GPL*, creado por Ludvig Strigeus.

Como su nombre indica, *ScummVM* ejecuta los juegos a través de una máquina virtual, usando solamente sus archivos de datos y reemplazando los ejecutables con los que el juego fue originalmente lanzado. Esto permite ejecutar los juegos en sistemas para los cuales nunca fueron diseñados y de ahí su gran portabilidad. Éste es un ejemplo claro de las facilidades que una máquina virtual puede dar para la portabilidad de las aplicaciones que sobre ella se pueden ejecutar, y actualmente el futuro de este sistema basado en una máquina virtual de propósito tan específico es precisamente ampliar el número de juegos y plataformas soportados. Esta máquina es soportada por múltiples plataformas como *Win32*, *Linux i386*, *PPC*, *Solaris* y otras.

6.4.3.11 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Todos los sistemas estudiados son sistemas de computación multiplataforma. Aunque en *Smalltalk* no era el objetivo principal, la independencia del sistema operativo y procesador *hardware* ha sido buscada en el diseño de todos los sistemas vistos. Las máquinas abstractas de *Smalltalk*, *Self* y *Java* fueron desarrolladas para dar un soporte portable a un lenguaje de programación, es decir, que su arquitectura está pensada especialmente para trabajar con un determinado lenguaje. Esta limitación no existe sin embargo en la plataforma *.NET*, capaz de trabajar con un número elevado de lenguajes que se diseñen para su máquina virtual, con la ventaja añadida de que dichos lenguajes serán interoperables.

La mayoría de estas máquinas están pensadas para resolver cualquier tipo de problema de carácter general, aunque otros tienen un propósito concreto. En cuanto a los grados de flexibilidad de estas plataformas, todas ellas poseen soporte para introspección, permitiendo que se pueda analizar cualquier objeto en tiempo de ejecución. Un conjunto de las máquinas vistas poseen acceso al medio; en *Smalltalk* se pueden añadir primitivas, pero para ello hay que modificar la implementación de la máquina virtual. En *.NET* es posible generar código en tiempo de ejecución y crear nuevas clases (programación generativa) pero limitado sólo a la creación de tipos y objetos nuevos, sin permitir modificar el código previamente existente. El mayor grado de flexibilidad lo consigue la modificación de la máquina *Self* llevada a cabo en el proyecto *Merlin* [Assumpcao93], donde mediante el uso de reflexión computacional se pueden modificar partes del funcionamiento de la máquina.

En lo referente al nivel de abstracción del modelo de computación de la plataforma, con la excepción de *Self,* la mayor parte poseen un modelo demasiado orientado al lenguaje de programación. Esto hace que la traducción desde otros sistemas sea compleja si éstos poseen un modelo de computación diferente y que la implementación de capacidades reflectivas pueda tener dificultades añadidas. *Self* se basa en un modelo basado en prototipos muy simple, uniforme y puede representar a multitud de lenguajes orientados a objetos, además de permitir hacer una implementación de capacidades reflectivas avanzadas más sencillamente [Wolczko96].

Por último, destacar que la mayoría de los sistemas presentados son máquinas virtuales exclusivamente pensadas para una ejecución eficiente de lenguajes estáticos

(.*NET*, *Mono*, *Java*), en contraposición con la máquina *Parrot*, cuyo fin es similar a las anteriores, pero centrándose principalmente en lenguajes dinámicos.

6.4.4 Máquinas Abstractas No Monolíticas

Todas las máquinas abstractas estudiadas en esta sección no poseen una arquitectura monolítica, es decir, no definen un sistema computacional invariable con una política y una semántica. Esto implicaría que el modo en el que llevan a cabo la computación de las aplicaciones y sus diferentes características es constante y en ningún momento puede modificarse. Este aspecto monolítico de las máquinas abstractas podría limitar la implementación de ciertos grados de flexibilidad, por lo que merece la pena explorar que ventajas podrían aportar estos otros sistemas. A modo de ejemplo, la máquina virtual de *Java* [Sun95], permite modificar sólo un conjunto limitado de sus características: el modo que carga las clases. Para hacer esta operación, debemos emplear un `ClassLoader`. Además, es posible también modificar el modo en el que restringe la ejecución de un código por motivos de seguridad, mediante un `SecurityManager` [Eckel00]. Ésta es una solución de compromiso para determinados casos prácticos, pero no define una arquitectura verdaderamente flexible ya que sólo se podrá modificar aquello que previamente se haya establecido como modificable.

6.4.4.1 VIRTUAL VIRTUAL MACHINE

Actualmente existen aplicaciones con multitud de funcionalidades implementadas que son más típicas de un sistema operativo, pero carecen de interoperabilidad con otras aplicaciones. Un entorno computacional orientado hacia la reutilización debería dar soporte para la interacción entre aplicaciones (interoperabilidad interna) y que sea accesible desde cualquier lenguaje, código o representación de datos (interoperabilidad externa) [Folliot98]. Éstos son los objetivos buscados en el diseño de *virtual virtual machine* (*VVM*) [Folliot97], una plataforma multilenguaje, independiente del *hardware* y sistema operativo, y que es extensible y adaptable, de forma dinámica, a cualquier tipo de aplicación.

Una aplicación que se vaya a ejecutar sobre *VVM*, tendrá asociado un "tipo", que es el lenguaje en el que ha sido codificada. Cada "tipo" o lenguaje de programación tiene asociada una descripción para la máquina virtual denominada "*VMlet*". Esta descripción contiene un conjunto de reglas que describen cómo traducir la aplicación, codificada en su propio lenguaje, a una representación interna común a todos los lenguajes del modelo de computación de *VVM*. La interoperabilidad interna del sistema (reutilización de código, independientemente del lenguaje en el que fue desarrollado) se consigue gracias a la traducción a un único modelo de computación. La arquitectura propuesta se muestra en la Figura 6.18.

El procesador virtual es la raíz básica del modelo de computación del sistema. Ejecuta un lenguaje interno al cuál se traduce cualquier lenguaje de programación utilizado para acceder a la plataforma. Sobre este procesador virtual se construye el sistema operativo virtual, que implementa rutinas propias de un sistema operativo, siendo éste independiente de la plataforma y del lenguaje. Las herramientas de lenguajes son un conjunto de librerías de programación que proporcionan funcionalidades propias de un conjunto de lenguajes. Su representación también es independiente de la plataforma y lenguaje, al haber sido desarrolladas sobre el procesador virtual.

La interoperabilidad interna se apoya en un módulo de seguridad, crítico para

asegurar la integridad del sistema. Todas las reglas que definen la seguridad del sistema están dentro de este módulo, pero las reglas propias de cada lenguaje se encuentran en su *VMlet*. Las aplicaciones codificadas para *VVM* son traducidas dinámicamente, utilizando su *VMlet*, a la representación interna de computación, lo que permite que aplicaciones desarrolladas en distintos lenguajes interactúen entre sí y que una aplicación pueda desarrollarse en distintos lenguajes de programación. *VVM* posee también una consola para poder administrar y acceder a la plataforma virtual desde el sistema operativo real utilizado.

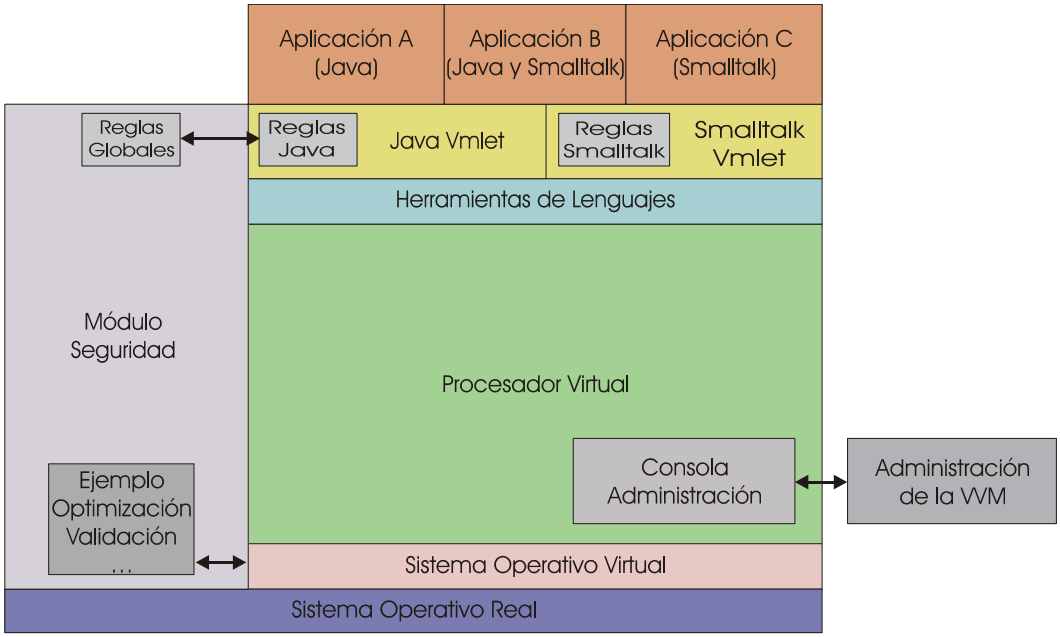

Figura 6.18: Arquitectura de *Virtual Virtual Machine*

6.4.4.2 *ADAPTIVE VIRTUAL MACHINE*

El diseño de "*Adative Virtual Machine*" (*AVM*) está enfocado a encontrar una plataforma de ejecución dotada de movilidad, adaptabilidad, extensibilidad y dinamicidad. Sobre ella, las aplicaciones podrán distribuirse de forma portable, segura e interoperable. Las aplicaciones se podrán adaptar a cualquier plataforma física, y se podrá extender su funcionalidad de un modo sencillo. Además, el entorno de ejecución de una aplicación podrá ser modificado de forma dinámica [Baillarguet98].

Se identifica el desarrollo de una máquina abstracta como mecanismo para lograr los objetivos mencionados, puesto que éstas ofrecen una separación entre las aplicaciones ejecutables y el *hardware* sobre las que se ejecutan [Baillarguet98]. Se diseña pues una máquina abstracta adaptable (*AVM*) sobre la que se ofrece un entorno virtual de ejecución (*Virtual Execution Environment* o *VEE*). En la Figura 6.19 se muestra la arquitectura del entorno virtual de ejecución.

AVM separa la parte estática, que representa el motor de ejecución o intérprete, de la parte dinámica que define las características propias del entorno de computación en la interpretación. La unión de ambas partes supone el entorno virtual de computación (*VEE*): un sistema de computación adaptable.

La parte dinámica son especificaciones de máquina virtual (*VMspec*): constituyen especificaciones de alto nivel de componentes de la máquina virtual, como sistemas de persistencia o planificación de hilos. Cada *VMspec* incluye las definiciones de ejecución y

un modelo de objetos, así como un conjunto de primitivas para acceder al sistema operativo. El compilador y optimizador de código transforman las *VMspecs* en una representación binaria denominada *VMlet*. El cargador de *VMlets* carga en memoria la especificación de la máquina virtual definida, y el generador de la máquina virtual la introduce en el sistema de interpretación adaptable (*AVM*). Una vez concluido este proceso, la máquina virtual se comporta con unas determinadas características hasta que sean modificadas con la carga de una nueva especificación. El intérprete de la máquina abstracta ha sido desarrollado como un *framework* orientado a objetos que produce un entorno de computación adaptable.

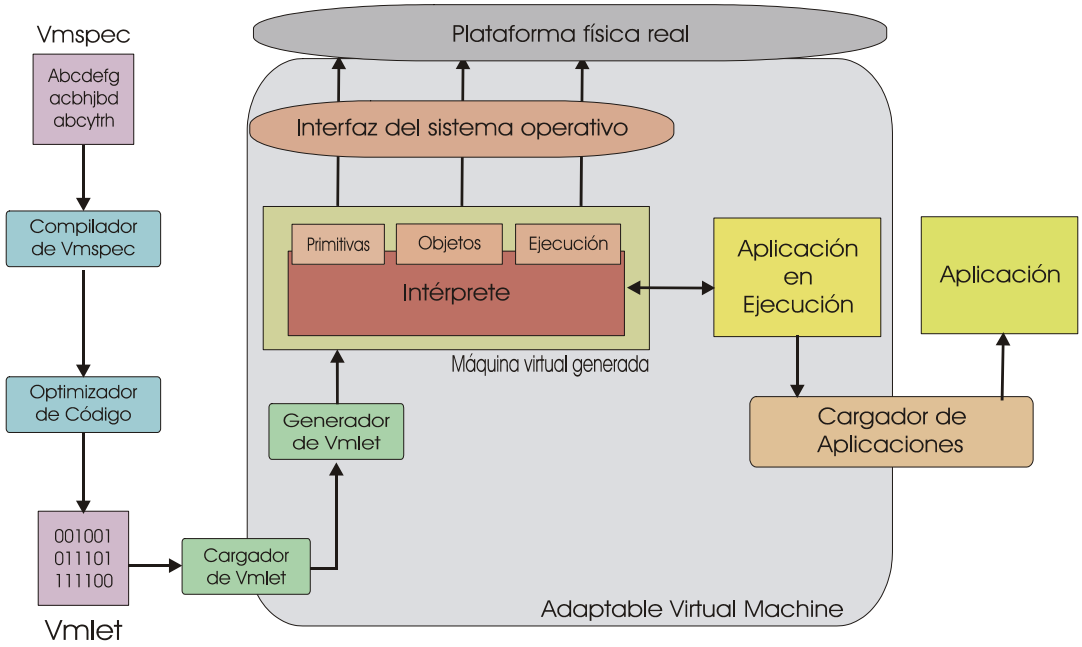

Figura 6.19: Arquitectura de la *Adaptable Virtual Machine*

6.4.4.3 *EXTENSIBLE VIRTUAL MACHINE*

La idea básica sobre la que se diseña *Extensible Virtual Machine* (*XVM*) [Harris99] es implementar un motor mínimo de ejecución dotado de una interfaz de acceso a bajo nivel, para poder diseñar una plataforma de ejecución de un amplio espectro de aplicaciones, adaptable a cualquier tipo de problema.

La mayoría de las máquinas abstractas existentes siguen un esquema monolítico, en el que la arquitectura de éstas posee un esquema fijo de abstracciones, funcionalidades y políticas de funcionamiento en la ejecución de sus aplicaciones. De forma contraria, un sistema extensible debería permitir a los programadores de aplicaciones modificar el entorno de ejecución, siempre en función de las necesidades demandadas por del tipo de aplicación que se esté desarrollando. Podemos poner la gestión de memoria *heap* como ejemplo: Determinados tipos de aplicaciones hacen un uso extenso de un tamaño concreto de objetos y establecen una determinada dependencia entre éstos. Para este tipo de aplicaciones, un modo específico de creación y ubicación de objetos en la memoria *heap* implica una reducción significativa en los tiempos de ejecución [Vo96].

Sceptre es el primer prototipo desarrollado para obtener *XVM*. Su diseño, orientado a la extensibilidad utiliza una fina granularidad de operaciones primitivas, mediante las cuales se desarrollarán las aplicaciones. Su arquitectura se basa en la utilización de componentes, implementando cada uno de ellos un conjunto de

operaciones. Una máquina virtual concreta se crea como composición de un conjunto de componentes. Al más bajo nivel, los componentes se implementan sobre la "*core-machine*" (motor base de ejecución), que proporciona acceso a la memoria y a la pila. Los servicios que ofrece un componente se especifican mediante una determinada interfaz denominada "*port*".

6.4.4.4 DISTRIBUTED VIRTUAL MACHINE

Distributed Virtual Machine (*DVM*) establece una arquitectura de servicios distribuidos que permite administrar y establecer un sistema global de seguridad para un conjunto heterogéneo de sistemas de computación sobre una misma máquina virtual [Sirer99]. Con *DVM*, características como verificación del código, seguridad, compilación y optimización, son extraídas de los clientes (máquinas virtuales) y centralizadas en potentes servidores.

En las máquinas virtuales monolíticas, como *Java* [Sun95], no es posible establecer un sistema de seguridad y una administración para el conjunto de todas las máquinas virtuales en ejecución. *DVM* ha sido diseñada e implementada partiendo de *Java*, descentralizando todas aquellas partes de su arquitectura monolítica que deban estar distribuidas para un control global a nivel de sistema. La funcionalidad centralizada de *DVM* se realiza mediante procesamiento estático (tiempo de compilación), y análisis y gestión dinámica (en tiempo de ejecución). El esquema se muestra en la Figura 6.20.

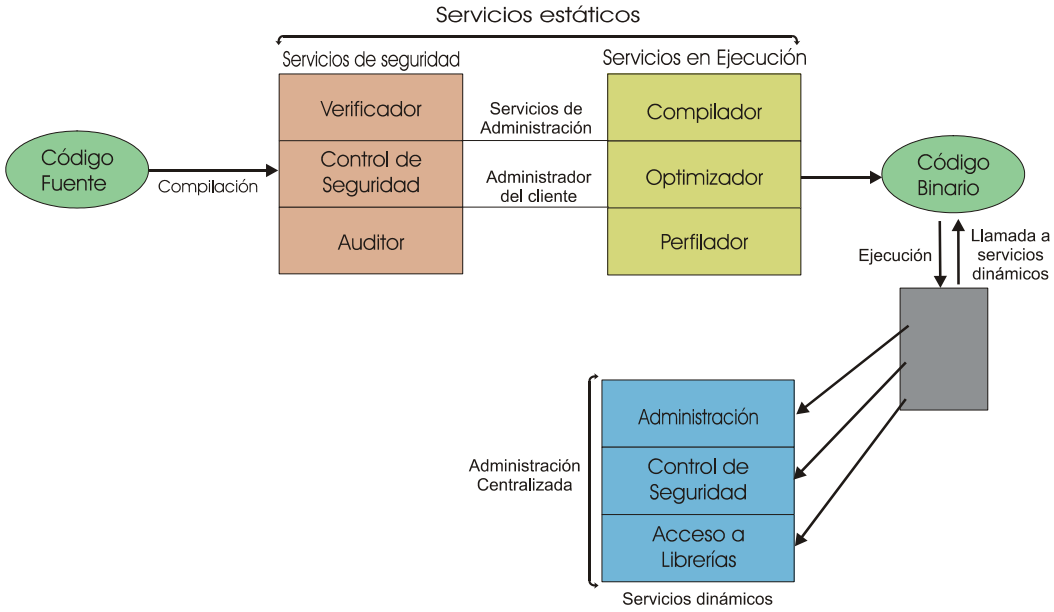

Figura 6.20: Fases en la compilación y ejecución de una aplicación sobre *DVM*

Los servicios estáticos que ofrece esta plataforma son: Verificador, compilador, auditor, perfilador y optimizador. Estos servicios analizan la aplicación antes de su ejecución y se aseguran de que cumpla todas las restricciones del sistema. Los servicios dinámicos complementan la funcionalidad de los servicios estáticos, proporcionando servicios en tiempo de ejecución dependientes del contexto del cliente. El enlace entre los servicios estáticos y dinámicos se produce gracias a un código adicional que se inyecta en los archivos binarios, en la fase de procesamiento estático.

Cuando en el análisis estático se encuentran operaciones que no pueden ser comprobadas en tiempo de compilación, se inserta en el código llamadas al

correspondiente servicio dinámico. Mientras que los problemas propios de una arquitectura monolítica son resueltos con *DVM* de una forma distribuida, ésta distribución puede llevar a una pérdida de eficiencia en tiempo de ejecución. La transmisión de información a través de la red es el principal cuello de botella con el que se podrían encontrar las aplicaciones, y se trata de solventar con técnicas como la incorporación de servidores que utilicen replicación de su información, por ejemplo.

6.4.4.5 APORTACIONES Y CARENCIAS DE LOS SISTEMAS ESTUDIADOS

Entre las características interesantes de las plataformas estudiadas, cabe decir que son multiplataforma y no dependen de forma directa de un lenguaje de programación. Además, todas están enfocadas a desarrollar una plataforma computacional de forma independiente a un problema preestablecido, aunque *DVM* está principalmente enfocado a la administración de una plataforma distribuida.

La flexibilidad y extensibilidad es la característica común que las agrupa. Rompen con las arquitecturas monolíticas típicas, y establecen mecanismos para modificar características del entorno de computación. Excepto *XVM*, todas ofrecen un mecanismo de extensibilidad basado en una arquitectura modular: Se identifican las funcionalidades susceptibles de ser modificadas y se establece una interfaz para extenderlas. En cambio, *XVM* ofrece una expresividad de su flexibilidad basada en un lenguaje de acceso a su "*core machine*", es decir, no se ofrece un conjunto de módulos variables, sino que el programador define una máquina virtual concreta apoyándose en la expresividad de un lenguaje.

No obstante, este diseño modular, aunque adecuado para conseguir un determinado nivel de flexibilidad que puede ser más o menos elevado, cuenta con el handicap de que solamente podrán flexibilizarse aquellas funcionalidades que previamente se identifiquen como tales, por lo que nunca se logrará una total libertad para hacer flexible cualquier parte del sistema, ya que si ésta no ha sido considerada como candidata para esto, no podrá hacerse. Además, este diseño modular hace que el tamaño del *framework* sea elevado, dificultando la implantación y migración de la plataforma a distintos sistemas tamaños de las distintas plataformas. Todo ello lleva a establecer la reflexión como el mecanismo más adecuado para aumentar la flexibilidad del sistema, al permitir, con una implementación adecuada, una flexibilidad potencialmente sin restricciones de que es y que no es flexible, así como una mayor sencillez conceptual.

6.4.5 Máquinas Abstractas para Protección de Código y la Propiedad Intelectual

6.4.5.1 STARFORCE

Describiremos a continuación el sistema de protección *Starforce* [Starforce06], famoso actualmente por ser probablemente el más difícil de desmantelar, consiguiendo un alto nivel de protección para el *software* al que se asocia. Puede afirmarse que la efectividad de este sistema es muy elevada, de manera que en muchas ocasiones sólo se

consigue desmantelar transcurridos varios meses, consiguiendo pues que cuando el producto que protege pueda usarse sin el soporte original que lo contiene gracias a que la protección se ha visto superada, éste haya perdido mucho interés en el mercado, al no ser una novedad (suele aplicarse en novedades en la industria del *software*, sobre todo en videojuegos). Otro efecto es que la forma de romper el sistema de protección suele conllevar operaciones que no están al alcance de un usuario estándar o bien son muy aparatosas (como la desconexión física de unidades de *CD-ROM IDE*).

Antes de describir este sistema, conviene destacar que no existe mucha información pública sobre la arquitectura del mismo, ya que una de sus grandes bazas es impedir a los usuarios externos que conozcan exactamente su funcionamiento y su código para dificultar aún más su ruptura.

Starforce, a partir de su versión 3, ejecuta la mayoría de su código sobre su propia máquina virtual. Esta máquina virtual posee una serie de instrucciones, y si se intenta emplear una herramienta para analizar sus instrucciones, de manera que se intente averiguar qué operaciones está ejecutando la máquina en un momento dado, lo que se obtendrá será una información inservible, ya que los datos de los *opcodes* de dicho lenguaje están encriptados de forma dinámica, es decir, el conjunto de instrucciones de la protección cambia de un *software* protegido a otro, aunque usen la misma versión. Esto básicamente significa que cualquier persona que desee averiguar cómo actúa la protección deberá emplear un gran número de cálculos (y un tiempo considerable) para intentar averiguar las instrucciones que se están ejecutando en un momento dado, ya que las instrucciones de esta máquina se pueden presentar de diferentes formas, por lo que habría que emplear técnicas heurísticas para averiguar de qué forma concreta se van a extraer para un único *software* dado. Otro efecto de esta característica de *Starforce* es que impide que se desarrollen herramientas genéricas que desprotejan *software* de manera general, lo que dificulta aún más esta labor (ya que habría que desarrollar una herramienta para cada producto protegido.

Como la protección se integra dentro de una parte del código del *software* que protege, al estar el código de la máquina encriptado y ser esta encriptación muy difícil de romper, el efecto conseguido es que el *software* original goce de muchas garantías de no ser usado ilegalmente. Esta dificultad, junto con otras técnicas relacionadas con la comunicación a bajo nivel entre *software* y *hardware* y la monitorización de ciertas partes del *SO* y de los dispositivos para interceptar determinadas operaciones, convierten a esta solución en una de las más efectivas.

6.4.5.2 TECNOLOGÍA DE PROTECCIÓN DE CONTENIDOS DE DISCOS *BLU-RAY*

El uso de máquinas virtuales también está presente como sistema de protección contra copias no autorizadas en uno de los últimos soportes digitales que ha aparecido en el mercado recientemente: *Blu-Ray* [CDRInfo06]. Este sistema, cuya competencia directa es el *HD-DVD*, usa la tecnología *AACS BD-ROM*, que incorpora, sobre la *AACS* "estándar" [Ayers06], nuevos elementos de seguridad, entre los que se encuentra la máquina virtual *BD+* que describiremos brevemente en esta sección.

En la siguiente figura podemos ver un esquema comparado general de la organización de elementos de seguridad en ambos formatos, donde vemos como *Blu-Ray* introduce la máquina virtual como una capa de seguridad adicional:

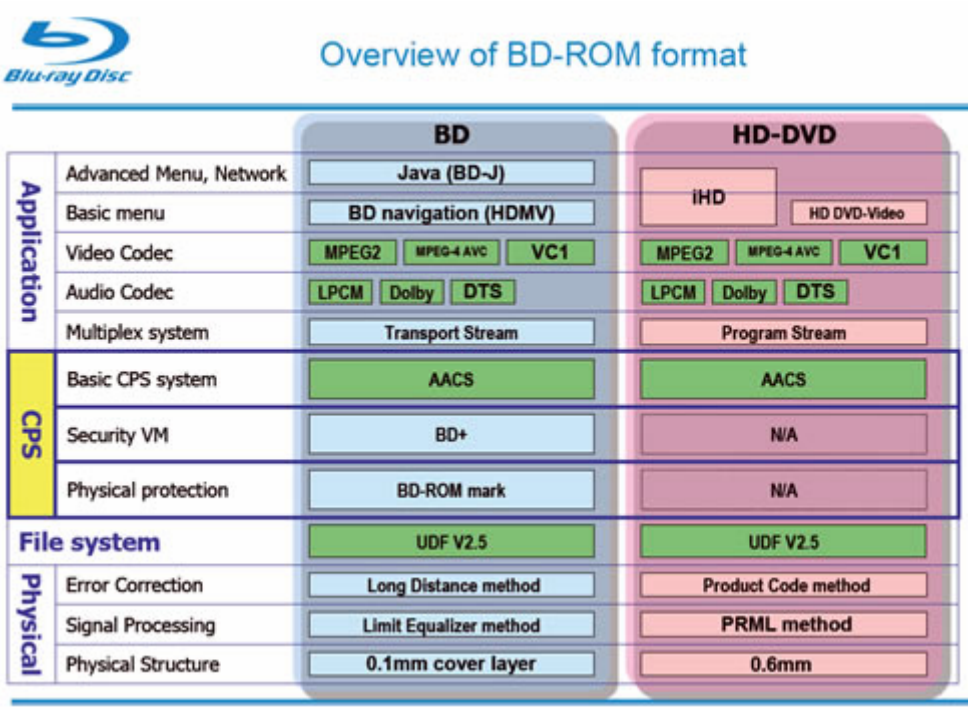

Figura 6.21: Organización de los elementos de seguridad en *Blu-Ray* y *HD-DVD* [CDRInfo06]

La tecnología de protección *AACS* permite proteger contenidos de audio y de video. Generalmente define las reglas que permitirán manipular dicho contenido y usa un sistema de encriptación avanzado con claves de 128 *bits*, entre otras tecnologías. Permite la anulación de la lectura de los contenidos en dispositivos que hayan sido manipulados y la autenticación avanzada de dichos dispositivos de lectura. Por último, también autoriza el soporte de copia en otros medios si así se desea.

Como se ve en el diagrama anterior, ambos medios tienen una estructura de seguridad organizada de forma muy similar, siendo el *Blu-Ray* el formato que introduce dos tecnologías adicionales: la máquina virtual *BD+* y la marca *BD-ROM*. La marca *BD-ROM* es una tecnología que impide grabar títulos en medios no autorizados. Se basa en un conjunto de datos "invisibles" que residen en el disco, y que impiden que aquellos medios de grabación que no tengan una determinado *hardware* puedan acceder al contenido del disco. Esta tecnología se apoya en la máquina virtual que describiremos a continuación.

BD+ es una pequeña máquina virtual que proporciona un entorno de procesamiento para la interpretación de código de contenido *BD+* (*BD+ Content Code*). Un material multimedia que esté en un disco puede incluir código específico para el mismo que será interpretado por la máquina. Durante la reproducción de un contenido, la máquina virtual ejecutará el código de contenido, aplicando comprobaciones de seguridad y permitiendo la reproducción del medio si el reproductor es legítimo (no ha sido alterado de manera no autorizada). Un código de seguridad se ejecutará continuamente durante la reproducción para corregir el flujo de datos de audio y de video y producir contenidos visibles, todo ello en tiempo real.

La siguiente figura ilustra la organización de los elementos que forman parte de esta tecnología y su funcionamiento general:

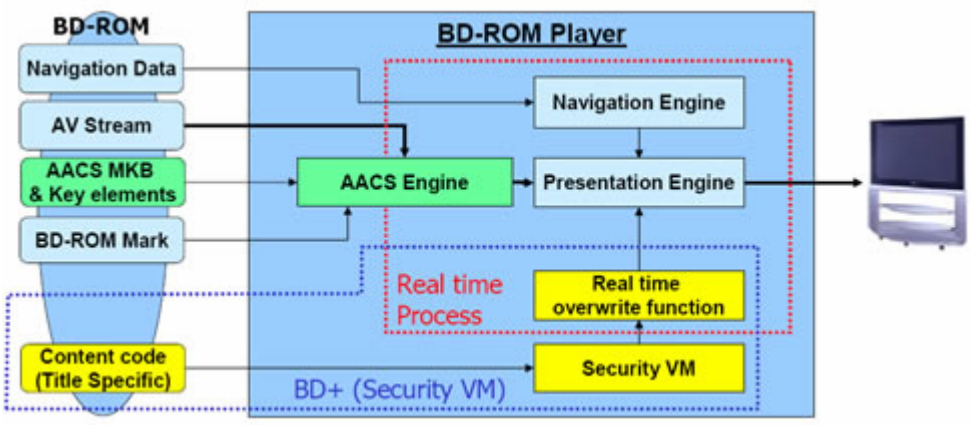

Figura 6.22: Organización de la máquina virtual BD+ [CDRInfo06]

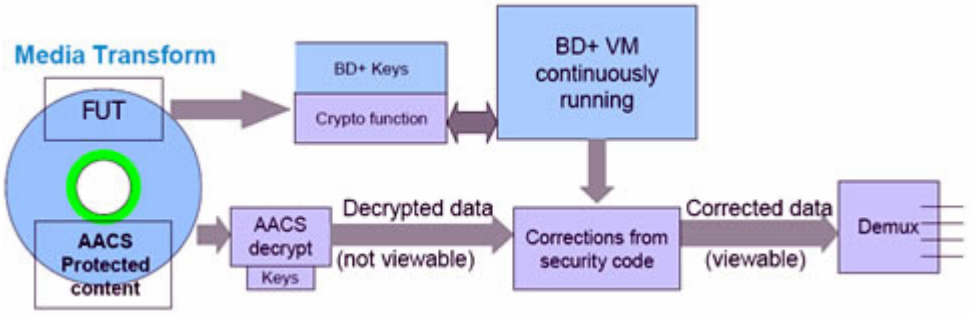

Figura 6.23: Funcionamiento de la máquina virtual BD+ [CDRInfo06]

El sistema de protección se basa en convertir los datos de audio y de video para que puedan ser vistos. Por ejemplo, parte de los datos en el disco pueden estar corruptos y no serán útiles a no ser que la máquina virtual *BD+* y el código de contenido que ésta ejecuta los corrija para poder visualizarlos correctamente. Esta máquina también permite detectar alteraciones en los reproductores de manera que un proveedor de contenidos pueda crear código de protección que responda a estas alteraciones, de forma que los discos sean capaces de responder ante alteraciones al *hardware* que se vayan produciendo. Para romper la protección ofrecida por este sistema el atacante debe, además de superar la protección del sistema AACS, romper el código de protección existente por cada uno de los títulos, lo que hace esta operación aún más compleja.

6.4.5.3 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas vistos intentan evitar la copia no autorizada de medios multimedia creando una infraestructura de seguridad compleja que se apoya en muchos factores diversos. No obstante, de cara a los objetivos de esta tesis, estos sistemas no nos son útiles dado su propósito específico y su diseño totalmente orientado a su principal y única función.

6.5 CONCLUSIONES

En este capítulo ha quedado claro que la utilización de una plataforma virtual basada en una máquina abstracta tiene múltiples ventajas, pero se busca principalmente la portabilidad de su código binario. Algunos sistemas desarrollados conseguían este objetivo para un determinado lenguaje, como *Java Virtual Machine*, mientras que otros sistemas, como la plataforma *.NET*, soportan multitud de lenguajes y no están vinculados a ninguno concreto, siendo ambos sistemas modernos y de gran éxito comercial.

La portabilidad del código ofrecida por una máquina virtual es explotada en entornos distribuidos de computación. Si el código de una plataforma puede ser ejecutado en cualquier sistema, éste puede ser fácilmente distribuido a través de una red de ordenadores. Además, al existir un único modelo de computación y representación de datos, las aplicaciones distribuidas en un entorno heterogéneo de computación pueden intercambiar datos de forma nativa, sin necesidad de utilizar una interfaz de representación común como en el caso de otros sistemas.

Además, hemos visto un tipo de plataformas de diseño más avanzado, que ofrecen programación portable, distribución de código, interacción de aplicaciones distribuidas y un modelo de computación único basado en el paradigma de la programación orientada a objetos. La implementación de aplicaciones es más homogénea y sencilla para entornos heterogéneos distribuidos. Cabe destacar en este sentido el proyecto *.NET* de *Microsoft*, que unifica sus sistemas operativos en una única plataforma basada en una máquina abstracta.

Uno de los principales inconvenientes de los sistemas mencionados es su arquitectura eminentemente monolítica, que ofrece un modelo estático de computación, careciendo de flexibilidad computacional, es decir, que las características computacionales de la plataforma no pueden modificarse sin hacer una profunda reestructuración de la arquitectura de la máquina. Las arquitecturas monolíticas requerirán pues una extensa modificación en caso de querer adaptarse para la ejecución de otros modelos computacionales, que debe ser planificada cuidadosamente para no incurrir en una penalización excesiva de rendimiento, o bien una ruptura de la compatibilidad con el código ya desarrollado.

En cambio, el desarrollo de plataformas no monolíticas se centra en la descomposición modular de sus características. Mediante la selección dinámica de estas funcionalidades se pueden obtener distintas máquinas virtuales, específicas para un determinado tipo de problema. Sus limitaciones principales son el tamaño y complejidad de los sistemas obtenidos y el establecimiento a priori de las características modificables, que limita la posibilidad de poder modificar algo que no haya sido identificado como tal en el diseño, y que no se podrá modificar posteriormente.

Por otra parte, el estudio de los sistemas presentados nos permite encaminar la selección de una máquina virtual como base para el trabajo a desarrollar. Por una parte, la creación de una máquina virtual nueva que sea comparable a las existentes en aspectos como rendimiento, consumo de memoria, funcionalidad, soporte para lenguajes de programación usados hoy en día y herramientas disponibles no entra dentro de los requisitos de esta tesis (necesitamos un entorno con proyección comercial) y sería una labor enormemente compleja. Por tanto, es necesario uno de los sistemas presentados tiene que ser usado como base para el trabajo a desarrollar.

Dentro de estos sistemas, los requisitos de independencia del lenguaje de programación, del problema y de la plataforma hacen que, de todos los sistemas presentados, sólo las plataformas independientes tengan un conjunto de características apropiado para ser elegidas como base para desarrollar este trabajo. Por otra parte, dada la proyección ya mencionada de que el sistema escogido debe tener hacia el desarrollo de aplicaciones comerciales reales, debemos contemplar aspectos como su uso actual

como plataforma de desarrollo, las herramientas disponibles para trabajar con las mismas, sus posibilidades de ampliación (acceso al código, etc.), librerías disponibles para la elección del sistema idóneo. Esto, junto con el requisito de que el sistema base tenga un alto rendimiento, y que permita incorporar nuevos lenguajes que sean interoperables con los existentes de manera sencilla, hace que finalmente nos decantemos por un sistema basado en el estándar *CLI* (como *.NET*).

No obstante, de los posibles sistemas a escoger finalmente según los requisitos establecidos, la elección recaerá en una máquina que, independientemente de sus características particulares, va a ser una máquina virtual "estática" (es decir, optimizada para la ejecución de lenguajes con un sistema de tipos estático), ya que son las únicas que actualmente cumplen con los requisitos establecidos. En un capítulo posterior detallaremos más las razones que justifican esta elección.

7 Optimización de Aplicaciones en Tiempo de Ejecución

7.1 Introducción

El empleo de técnicas de optimización de aplicaciones en tiempo de ejecución es una parte muy importante del trabajo desarrollado, ya que queremos lograr aumentar la eficiencia de los lenguajes dinámicos que se ejecuten sobre la máquina virtual que escojamos, y estas técnicas son el medio para ello. Este capítulo describirá las técnicas de optimización dinámica de código más importantes empleadas actualmente, mostrando su estructura y sus bases de funcionamiento.

Aunque se presentarán en las siguientes secciones diversas técnicas de optimización, el empleo de una de ellas no es excluyente, es decir, es posible (como de hecho ocurre en sistemas actualmente en funcionamiento [SunHotSpot2006]) emplear varias técnicas al unísono, o bien partes de las mismas, para lograr una optimización de código adecuada a las características del sistema base sobre el que se aplican, no siendo por tanto técnicas cerradas que excluyan el uso de otros métodos.

Por último, ha de tenerse en cuenta que en este capítulo nos referiremos en todo momento a técnicas de optimización dinámica de código, que entrarán en marcha principalmente durante la ejecución de un programa concreto. No son objeto de estudio en este capítulo técnicas de optimización de código de alto nivel estáticas [EventHelix06] o bien aquéllas que un compilador posea y aplique cuando efectúe el proceso de compilación del código [Nullstone06]. Éstas se aplicarán o no en función de las características concretas del programa, de la herramienta de compilación y de la información acerca del programa y el desarrollo de su ejecución que esté disponible en un momento dado.

7.2 Técnicas de Optimización Dinámica de Aplicaciones

7.2.1 Compilación JIT

La compilación *JIT*, es una técnica que data de los años 60, pero que ha logrado mucha popularidad recientemente gracias a la incorporación de la misma a plataformas

ampliamente difundidas y utilizadas, como *Java*. Los sistemas de compilación *Just In Time* (o *JIT*) son creados con el propósito de mejorar los tiempos de ejecución de los programas de bajo nivel de abstracción ejecutados por una máquina virtual y también la memoria ocupada por los mismos. Para ello, esta técnica trata de obtener las ventajas de la compilación estática y de la interpretación de programas, evitando en lo posible los inconvenientes de ambas [Aycock03]. En un sistema que emplee esta técnica de compilación, el código de un programa es traducido a código nativo durante su ejecución, aunque este proceso de traducción suele llevar asociado otras acciones intermedias:

- El código de alto nivel se compila precisamente a una forma intermedia de código denominada *bytecode*, que ofrece una mejor portabilidad y capacidad de optimización.
- Este *bytecode* es el que posteriormente es traducido a código nativo mediante diferentes procedimientos, entre los que podemos destacar la compilación de funciones sólo en el momento en el que van a ser ejecutadas (de ahí el nombre de esta técnica, al compilarse las funciones "justo a tiempo"), asegurándose así pues que sólo se procesará el código que realmente es empleado durante una ejecución concreta del programa.

La compilación *JIT* por tanto combina lo mejor de ambas técnicas de procesamiento de código: los mecanismos básicos de optimización y todas las operaciones más costosas asociadas al procesamiento del código fuente se realizan en tiempo de compilación a *bytecode*, mientras que la transformación de *bytecode* a código máquina en tiempo de ejecución es mucho más eficiente que si se partiese del código de alto nivel. Otra de las ventajas obtenidas es que el código intermedio generado es portable (es una forma intermedia, no asociada a ninguna arquitectura en particular) y mucho más simple, por lo que para lograr la ejecución del programa en otra máquina sólo sería necesario desarrollar un compilador adecuado de una complejidad menor.

Por tanto, esta técnica está claramente diferenciada de las que se emplean tradicionalmente para la traducción de programas: compilación e interpretación. En ambos casos los programas se crean en un lenguaje de alto nivel cualquiera y son posteriormente traducidos a código directamente ejecutable por la máquina. La compilación efectúa la traducción de un lenguaje a otro y puede constar de varias fases, pudiendo traducirse el programa a diferentes lenguajes intermedios hasta llegar a un lenguaje directamente ejecutable por la máquina. La interpretación hace el mismo análisis del programa que la compilación, pero ejecuta en el mismo momento las instrucciones que va procesando, eliminando pues cualquier paso intermedio entre lenguajes. Las características principales de ambas técnicas son:

- Los programas compilados suelen ejecutarse más rápido, especialmente si la compilación es a un lenguaje directamente ejecutable por el *hardware* subyacente. Este tipo de compilación puede emplear un determinado tiempo variable para analizar y optimizar el programa y su código, pudiendo pues graduar el nivel de optimización que se va a aplicar al mismo a costa de aumentar el tiempo empleado en compilar el programa. En la compilación *JIT*, emplear un tiempo excesivo en la optimización del código nativo generado durante la ejecución de un programa puede causar pausas en la misma, que podrían no ser aceptables para los usuarios.
- Los programas interpretados ocupan normalmente menos espacio en memoria, al ser la representación de su código de más alto nivel y poder llevar mucha más información semántica implícita que un código de bajo nivel.
- Los programas interpretados suelen ser más portables, gracias a técnicas como las

descritas anteriormente.

- Los intérpretes tienen acceso a una variada información en tiempo de ejecución, que puede cambiar entre ejecuciones incluso de un mismo programa, y que puede no estar disponible en tiempo de compilación. Esta información puede ser desde información relativa a los tipos usados hasta incluso aspectos específicos del sistema o máquina sobre el que se ejecutan. Poseer esta clase de información adicional puede tener una serie de beneficios, como mejorar ciertos aspectos relativos al procesamiento del código del programa (al haber más información disponible se podrán tomar más decisiones) o ejecutar algoritmos que en tiempo de compilación sea imposible aplicar por carecer de la información necesaria, y que puedan tener un efecto beneficioso global en algún aspecto del programa. Esto también incluye la posibilidad de adaptar el código nativo generado a las características concretas del sistema que lo ejecuta o a su situación particular en un momento dado, pudiéndose lograr una mejora de rendimiento al ser el código nativo generado más afín al sistema concreto sobre el que se ejecuta.

7.2.1.1 CLASIFICACIÓN DE SISTEMAS JIT

Es posible hacer una clasificación de sistemas *JIT* de acuerdo con tres criterios comunes a todos ellos [Aycock03]:

- **Invocación**: Un compilador *JIT* se llama explícitamente si el usuario debe realizar alguna acción concreta para ejecutar el compilador en la ejecución de un programa. En caso contrario (invocación implícita) el uso de ese compilador es completamente transparente al usuario. Existen numerosos ejemplos de compiladores *JIT* implícitos, como son los incluidos en las plataformas *Java* y *.NET*. Un compilador explícito es por ejemplo el poseído por el sistema *HiPE* (*High Performance Erlang*) [Johanson00].
- **Ejecutabilidad de lenguajes**: Los sistemas *JIT* normalmente tienen asociados dos lenguajes: el lenguaje origen desde el que traducir y el lenguaje destino al que deben generar el código. Estos dos lenguajes pueden ser el mismo si el sistema está realizando optimizaciones en plena ejecución. Un sistema *JIT* se denomina monoejecutable si puede ejecutar sólo uno de estos lenguajes, mientras que se llama poliejectuable si puede ejecutar más de uno. Un sistema de este último tipo puede decidir cuando se invoca al compilador, pudiendo por tanto emplear la representación del programa que desee en cada momento. No obstante, más que una característica propia de los compiladores *JIT*, ésta es una característica del diseño de los compiladores [Aycock03].
- **Concurrencia**: Esta propiedad determina cómo se ejecuta el compilador *JIT* respecto al propio programa que está en funcionamiento. Si la ejecución del programa se suspende para permitir la compilación (como en el caso de *Java*), entonces el compilador no es concurrente, ya que el compilador en este caso se ejecutará mediante la invocación de una rutina, la transmisión de un mensaje u otro mecanismo. Un compilador concurrente (presente en trabajos realizados con el lenguaje *Oberon* [Kistler97]) funciona en un proceso o hilo separado al del propio programa, pudiendo incluso ser asignado a un procesador diferente, en un proceso muy similar a la compilación continua.

En la actualidad predominan los compiladores *JIT* implícitos. Los compiladores concurrentes están en constante desarrollo y se esperan avances en el futuro. El uso de compiladores *JIT* sobre máquinas virtuales actualmente permite a los lenguajes que

pueden servirse de ella una ganancia de rendimiento importante. A modo de ejemplo, se han establecido comparaciones entre el rendimiento de un mismo lenguaje (*Java*) interpretado y compilado mediante *JIT* para ver la ganancia de rendimiento existente con esta última técnica [Hardwick97]. Estos estudios revelan que el uso de *JIT* hace que la mayoría de operaciones se ejecuten de 5 a 30 veces más rápido que con un intérprete, lo que es una ganancia de rendimiento global muy importante, si bien no es la misma en todas las posibles operaciones.

7.2.2 Compilación Continua:

La técnica de compilación continua se basa en la coexistencia en memoria de un intérprete y un compilador para un mismo programa [Plezbert96]. El intérprete será el encargado de la ejecución del programa, mientras que el compilador traduce el código fuente a código nativo directamente ejecutable por el sistema. Ambos módulos se ejecutan concurrentemente (tanto si realmente hay concurrencia, por existir múltiples procesadores en el sistema, como si es concurrencia aparente). La figura 7.1 muestra la arquitectura de un sistema de estas características:

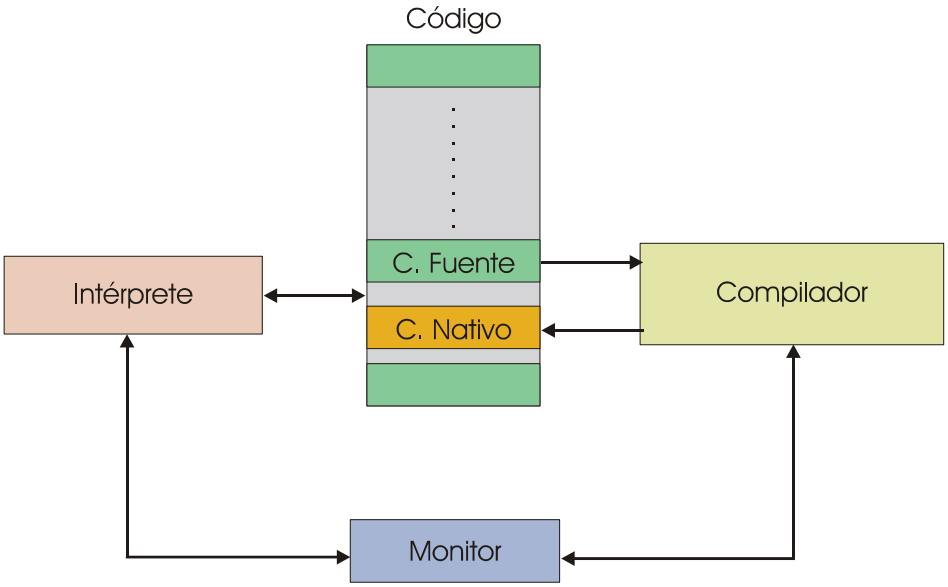

Figura 7.1. Esquema de un compilador continuo

Los elementos que aparecen en esta figura, y que servirán para describir el modo de funcionamiento concreto de esta técnica, son:

- **Código**: El código del programa en un momento dado estará compuesto por una mezcla de versiones del mismo. Parte del código será código fuente (destinado a ser interpretado) y otra parte estará ya compilado a código directamente ejecutable por el sistema subyacente. Antes del inicio del mismo, todo el código "de usuario" del programa se encontrará en forma de código no nativo (las librerías que el programa emplee se supone que ya estarán traducidas a código nativo). A medida que el programa se ejecuta, el compilador traduce a código nativo las funciones, de manera que hasta que una función no está completamente traducida a código nativo, el intérprete no podrá hacer uso de la versión nativa (y

por tanto más eficiente) de la misma.

- **Compilador**: Cuya función acabamos de describir.
- **Intérprete**: Módulo responsable de la ejecución del programa siguiendo las pautas descritas: procesará el código de la aplicación empleando las versiones compiladas a código nativo de las funciones del programa a medida que estén disponibles. Para ello hay dos posibles estrategias que veremos más adelante.
- **Monitor**: Es el módulo empleado por el compilador y el intérprete para comunicarse entre sí. Esta estructura de datos es compartida y contiene la información necesaria por ambos módulos para intercambiar la información necesaria para sus tareas. Un ejemplo de ello es la construcción por parte del intérprete de información acerca del programa a medida que transcurre su ejecución, que podrá ser empleada por el compilador para realizar de una forma más sencilla sus labores, información a la que podrá acceder a través del monitor.

7.2.2.1 COMPARACIÓN CON MÉTODOS TRADICIONALES

Al igual que la compilación *JIT*, el modelo de compilación continua trata de combinar los beneficios de la interpretación de código con los de la compilación. Con un compilador de este tipo se trata de obtener todos los beneficios más significativos de la interpretación de código (incluyendo la independencia de la plataforma y el menor tamaño del archivo del programa), mientras que, si el compilador se configura de manera que emplee un tiempo elevado en la optimización del código que genera, el sistema derivará a medida que se vaya ejecutando en un programa altamente optimizado y completamente compilado a código nativo, que potencialmente pueda tener un rendimiento mejor que el producido por un compilador estático "tradicional". Por otra parte, una diferencia sustancial de esta técnica con la compilación *JIT* es que permite que los programas comiencen a ejecutarse antes.

7.2.2.2 ESTRATEGIAS DE COMPILACIÓN

Según [Plezbert96] el rendimiento de un programa que emplee un compilador continuo se ve muy afectado por el orden en el cual las rutinas se traducen a código nativo. Para ello existen múltiples estrategias:

- ***Shortest-first***: Esta estrategia se basa en el tamaño de los módulos de código. Los módulos se ordenan de menor a mayor y son traducidos en ese orden.
- ***Largest-first***: Esta estrategia es la contraria a la anterior.
- **Aleatorio**: Los módulos son escogidos para su traducción a código nativo de manera aleatoria. El propósito de esta estrategia es emular el no determinismo que puede causar la transmisión por red de módulos de código para ser compilados, escenario en el que los módulos pueden no llegar en el orden enviado.
- **Más frecuentemente ejecutado hasta el momento**: El monitor en esta estrategia amplia sus funciones para llevar una contabilidad de la frecuencia de uso de cada rutina. Cuando se necesite seleccionar un módulo para traducir, el compilador escogerá la rutina no traducida que acumule el mayor número de llamadas hasta ese momento de la ejecución del programa.
- **Más tiempo empleado hasta el momento**: Esta estrategia es muy similar a la anterior, pero se usa como medida el tiempo empleado en la ejecución de la rutina

(sin incluir el tiempo empleado en la llamada a otras rutinas desde dentro del código de ésta) en vez del número de llamadas.

- **Más frecuentemente ejecutado en global**: Esta estrategia emplea información recogida en una ejecución anterior del programa para determinar qué rutinas fueron las más llamadas durante la ejecución completa del mismo. El código no traducido que haya recibido el mayor número de llamadas durante la ejecución anterior será el elegido para ser traducido.
- **Más tiempo empleado en global**: Idéntica a la anterior, pero basándose en información recogida acerca del tiempo de ejecución.

Tres de estas estrategias (*shortest-first*, *largest-first* y aleatorio) son estáticas y usan información estática (en el sentido de que no es recabada durante la ejecución del programa y el orden de traducción se determina antes de la ejecución del mismo) para hacer los cálculos pertinentes. Otras dos estrategias (más frecuentemente ejecutado y más tiempo empleado) son dinámicas y usan información recabada al monitorizar la ejecución de un programa para determinar el orden de traducción. El resto de estrategias son hibridas, ya que el orden de traducción se escoge antes de la ejecución del programa, pero sin embargo necesitan de conocimiento obtenido durante la propia ejecución, aunque se obtiene a partir de una anterior a la actual.

Las tres estrategias estáticas tienen la ventaja de tener un impacto pequeño en el rendimiento del programa. El orden de traducción se determina antes de que comience la misma y los datos necesarios para ello se pueden obtener y consultar fácilmente del código fuente. Las estrategias dinámicas en cambio tienen un coste mayor, determinado por los cálculos necesarios para su aplicación y que el orden de traducción puede cambiar a medida que el programa se ejecuta. Las últimas dos estrategias requieren de información obtenida de una ejecución anterior que sea fiable para la ejecución actual, algo que no siempre puede ser posible.

La compilación puede tener lugar a nivel de archivo, examinando las estadísticas de cada archivo en conjunto y no las de cada rutina. La estrategia de "más frecuentemente ejecutado hasta el momento" emplearía el número de llamadas usado para todas las rutinas definidas en cada archivo, por lo que es posible que finalmente no se compile la rutina realmente más empleada primero. No obstante, ambas estrategias (a nivel de archivo y a nivel de rutina) son igualmente válidas para estas técnicas.

Por último, es importante mencionar que, si se supone que el tiempo requerido para la compilación de un archivo es una función que depende del tamaño del archivo a compilar, entonces la estrategia *shortest first* maximizaría la productividad del compilador, aunque esto no correspondería necesariamente con una maximización de su rendimiento.

7.2.2.3 ESTRATEGIAS DE REEMPLAZO

La estrategia de reemplazo es usada por el intérprete para reemplazar el código fuente por su equivalente traducido a código nativo. Según los estudios llevados a cabo en [Plezbert96] esta estrategia también tiene un impacto considerable en el rendimiento en ejecución de la aplicación. Aunque se pueden definir múltiples estrategias de reemplazo, se consideran dos de ellas como más representativas:

- **Reemplazar en la llamada**: En esta estrategia, el cambio entre código interpretado a código nativo ocurre únicamente en las llamadas a funciones. Cuando ocurre una llamada a una función, se hace un salto al código nativo si está

disponible, interpretándose la función en caso contrario. Cuando se termina de ejecutar la función, la ejecución del código que la llama continúa en el modo en el que estuviese antes de la llamada a la misma. Por ejemplo, si una rutina interpretada A llama a otra B que está ya traducida a código nativo, cuando B termine se devuelve el control a la rutina A, que continua siendo interpretada. Por tanto, las invocaciones que están activas en la pila de ejecución en un momento dado no son reemplazadas si una nueva versión compilada de la función está disponible en ese preciso momento. Nótese que esto no ocurre en caso de llamadas recursivas, donde es posible el empleo de la función nativa en las llamadas posteriores a su compilación.

- **Reemplazo preemptivo**: En esta estrategia la rutina se cambia de su versión interpretada a la compilada en el momento en el que está disponible, incluso si la rutina original está en plena ejecución. Además, si la versión en código nativo está disponible justo cuando la función se está sacando de la pila de ejecución para reanudar su ejecución, entonces se hace uso igualmente de su versión nativa, incluso aunque esta función fuese interpretada cuando se apiló originalmente.

La primera estrategia tiene la ventaja de tener un bajo coste y ser fácilmente implementable. Cuando el intérprete ejecuta la llamada a la función, comprueba la información del estado del programa para ver si está disponible la función equivalente en código nativo, empleando dicha función si se encuentra. Por otra parte, si la función que se está traduciendo tiene llamadas a funciones que aún no han sido traducidas, estas llamadas seguirán siendo interpretadas.

La estrategia preemptiva en cambio es más difícil de implementar y tiene un coste mayor, pero puede mejorar el rendimiento significativamente comparada con la anterior. Para poder ejecutar correctamente esta estrategia, es necesario incluir código en el intérprete que compruebe la disponibilidad de nuevo código nativo cada vez que se elimine un registro de activación de la pila para continuar la ejecución, entre otras operaciones. También es necesario que el compilador pueda interrumpir al intérprete para poder completar las operaciones planteadas, pero esto puede tener un coste prohibitivo, que hace que no se lleve a cabo en algunas implementaciones.

Como hemos visto, el modelo propuesto por la compilación continua tiene los beneficios de la interpretación de código mientras proporciona grandes avances en términos de rendimiento, llegando incluso a poder ser teóricamente superior a la compilación tradicional de programas [Plezbert96b], al poder emplear un nivel mayor de optimización o la habilidad de dividir efectivamente el procesamiento del código en dos procesos independientes. Por otra parte, el rendimiento de la aplicación dependerá del diseño de varios aspectos del compilador, como las diferentes estrategias de selección de código. A modo de resumen, las ventajas esperadas de un modelo de compilación continua son:

- Tiempo de respuesta inmediato, equivalente al de un intérprete.
- Un tiempo de recuperación más pequeño que la compilación tradicional. Se entiende por tiempo de recuperación el tiempo necesario para alcanzar un punto concreto fijado en la ejecución de un programa a partir de su código inicial no compilado. Para un compilador "tradicional", este tiempo incluye el tiempo necesario para compilar el programa, enlazarlo y luego el que se emplee en llegar al punto de ejecución mencionado. La estructura de estos compiladores hace que este tiempo sea menor [Plezbert96b], debido a que no tiene que pasar por una fase previa de compilación estática.
- Rendimiento en ejecución que se acerca al de la compilación tradicional a medida que transcurre la ejecución del programa [Plezbert96b].

- Ficheros a distribuir más pequeños.
- Distribución de aplicaciones en un formato independiente del sistema destino.
- Se puede emplear más tiempo en la optimización de código sin que el usuario tenga que esperar a que el proceso de compilación finalice.

No obstante, aún es necesario la implementación de un sistema que soporte este paradigma de propósito general que goce de una popularidad y utilidad equivalente a los sistemas *JIT* usados actualmente en plataformas como *.NET* y *Java* [SunHotSpot2006].

7.2.3 Optimización Adaptativa:

7.2.3.1 PRINCIPALES CARACTERÍSTICAS

Esta técnica está siendo también utilizada para lograr incrementos significativos de rendimiento actualmente, con varias líneas de investigación abiertas [Cooper05]. Su funcionamiento consiste en la existencia de un compilador adaptativo, que funciona en un ciclo constante de compilación-ejecución-análisis para encontrar el conjunto de optimizaciones y parámetros que permitan alcanzar ciertos objetivos de rendimiento, como puede ser el espacio ocupado en memoria por un programa o su rendimiento en ejecución, adaptándose a las características del programa (de ahí su denominación). Aunque este método se ha probado efectivo para estas tareas, la compilación adaptativa posee dos problemas principales:

- Es necesario un tiempo considerable para hacer las múltiples compilaciones y ejecuciones necesarias para realizar la labor del compilador de la forma más óptima posible.
- El mecanismo inherente en el ciclo que debe describir un compilador de este tipo es muy complejo, lo que no facilita ni su implementación ni su uso efectivo.

El proceso seguido por esta técnica en el ciclo mencionado consiste en analizar continuamente el rendimiento del programa y su código, en búsqueda de "*hot spots*", denominándose así aquellas porciones de código que son ejecutadas de forma más frecuente o repetitiva. Estas porciones de código se marcan para ser optimizadas, incrementando pues el rendimiento del programa final sin imponer teóricamente un coste excesivo, al no procesar aquel código que no tiene tanta importancia de cara al rendimiento. Dado que esta técnica puede tener multitud de variantes, y la optimización puede efectuarse incluso en varias pasadas para lograr niveles de optimización mayores según la frecuencia con la que se ejecute el código, la mayor parte del tiempo empleado en el proceso de compilación adaptativa se consume en procesar múltiples veces el código que se compila. Existen técnicas desarrolladas para minimizar el coste de este proceso, como por ejemplo la ejecución virtual [Cooper05], mediante la cual el programa se ejecuta una sola vez, y en dicha ejecución única se recopila una determinada información que permita predecir y estimar el rendimiento derivado de aplicar diferentes secuencias de optimizaciones sin necesidad de volver a ejecutar el código del mismo.

Otro problema que esta técnica plantea es determinar el conjunto correcto de optimizaciones para toda la variedad de posibles códigos fuente que pueda procesar el

compilador, siendo esto un proceso muy costoso que muchas veces requiere sistemas especializados en esta tarea [Kulkarni03]. Las mejoras en el *hardware* disponible han permitido actualmente explorar diferentes combinaciones de secuencias de optimización [Cooper99] [Cooper01], aunque estos experimentos han demostrado que la efectividad de una secuencia de optimizaciones para mejorar el rendimiento depende enormemente del código ejecutado.

Por tanto, la única forma que por el momento parece viable para solventar este problema es que sea el compilador el que pueda cambiar su comportamiento para cada programa de entrada. La adaptación podría hacerse de dos formas:

- Analizando el código de entrada para detectar ciertas características sobre las que se sepa que se pueden aplicar determinadas optimizaciones con un buen resultado [Whitfield97] [Zhao2003]. No obstante, la determinación de secuencias de optimizaciones óptimas para un código concreto es una labor muy compleja que no siempre puede aplicarse efectivamente a un programa concreto.
- Empleando un método en varias fases, de manera que un compilador adaptativo, al que se le proporcionen una serie de datos iniciales, compile el código con una determinada secuencia de optimizaciones. El sistema posteriormente ejecutará el código y medirá el rendimiento del mismo, evaluándolo, y ordenando al compilador su recompilación usando una secuencia modificada de optimizaciones en función de los datos obtenidos hasta el momento, repitiéndose el proceso. En este caso, múltiples técnicas pueden usarse para guiar al compilador en este proceso, como algoritmos genéticos y muestreo aleatorio, entre otros. Este método suele encontrar soluciones bastante óptimas, aunque sigue teniendo un coste en ejecución prohibitivo en cuanto el programa alcanza un determinado tamaño.

Por tanto, es necesario un método que reduzca el coste de estas operaciones de optimización para que esta técnica sea viable, lugar donde se ubica la ejecución virtual antes mencionada [Cooper05].

7.2.3.2 IMPLEMENTACIÓN DE COMPILADORES ADAPTATIVOS: *JVM*

Una vez expuestas las características principales de esta técnica de optimización, veremos ahora una implementación de la misma existente en un sistema comercial. La máquina virtual *Java* (*JVM*) posee optimización adaptativa de forma combinada con técnicas de compilación *JIT* [SunHotSpot2006]. La combinación de ambas técnicas está pensada para aprovecharse del principio de localidad espacial y temporal que todo programa posee [Denning05]. Dado que, en la mayoría de los programas, sólo una pequeña parte de su código se ejecuta durante la mayor parte del tiempo que están en ejecución, la máquina virtual *Java Hotspot VM* ejecuta cada programa empleando un intérprete mientras analiza el código que se está ejecutando para detectar los *hotspots* descritos anteriormente, todo ello con la idea ya mencionada de dedicar más tiempo para la optimización de aquellas partes del programa que se consideren críticas en cuanto a rendimiento. La monitorización descrita en este sistema continua mientras el programa está en ejecución, por lo que podría adaptar el rendimiento del programa en ejecución dinámicamente de acuerdo con las acciones del usuario.

El beneficio obtenido por este sistema combinado no sería posible si no se retrasara la compilación hasta que el programa esté en funcionamiento, lo que permite recabar información acerca de cómo se está usando el código en cada ejecución concreta y emplearla para hacer optimizaciones aún mejores. También se reduce el espacio ocupado en memoria por el código del programa. En concreto, la posibilidad de recabar

más información permite la aplicación de las siguientes técnicas:

- ***Inlining* de métodos**: En el lenguaje *Java*, la alta frecuencia de ejecución de métodos virtuales puede suponer un cuello de botella en cuanto a rendimiento. Gracias a la información recabada durante la ejecución, el sistema puede hacer estos métodos *inline* para mejorar así su rendimiento, ya que se reduciría la frecuencia de invocación dinámica de métodos y por tanto el tiempo empleado para esta operación. Las funciones *inline* producen bloques de código más grandes en los que el optimizador podrá trabajar, pudiendo así incrementarse la efectividad de las optimizaciones tradicionales introducidas por el compilador.
- **Deoptimización dinámica**: La técnica de *inlining*, aunque efectiva, puede tener una difícil implementación en el compilador, principalmente debido a la capacidad que tienen lenguajes como *Java* para cargar nuevo código dinámicamente durante la ejecución de un programa. Dado que ese nuevo código puede cambiar cómo se aplica el *inlining* en el programa, al poder devolver nuevos resultados el análisis que se lleva a cabo del mismo (el nuevo código puede alterar el funcionamiento del programa de formas impredecibles), el compilador debe ser capaz de "deoptimizar" dinámicamente (y reoptimizar si es necesario) aquellos *hotspots* optimizados anteriormente, para poder así efectuar *inlining* de forma segura.
- **Eliminación de código muerto**: [Nullstone06] Esta técnica consiste en la eliminación de código que no es alcanzable en la ejecución del programa o que no afecta a la misma.
- **Extracción de invariantes en bucles**: [Nullstone06] Toda expresión dentro de un bucle cuyo resultado no varíe puede ser trasladada fuera del bucle. La ganancia de rendimiento consiste en que entonces esta expresión sólo se ejecutará una vez, en lugar del número de iteraciones del bucle.
- **Eliminación de subexpresiones comunes**: [Cocke70] Consiste en buscar expresiones que se evalúen todas al mismo valor, y analizar si es conveniente reemplazarlas por una única variable que guarde su resultado (el valor en cuestión).
- **Propagación de constantes**: [Nullstone06] Si a una variable se le asigna un valor constante, ese valor puede propagarse por el código que emplee esa variable. La idea es sustituir la variable por el valor para hacer directamente los cálculos donde dicha variable intervienen (ya que es conocido su valor) y de esta forma reducir el tiempo empleado en evaluar expresiones en tiempo de ejecución, ya que todos o parte de los cálculos que debe llevar a cabo ya se habrán realizado.
- **Eliminación de comprobaciones de `null`**: Consiste en la localización y eliminación de comprobaciones de `null` que se juzguen innecesarias en el código, al determinarse que es imposible que la variable valga `null` en el momento de la comprobación.
- **Eliminación de comprobaciones de rango**: [Sun98] La especificación del lenguaje de programación *Java* requiere que los límites de los *arrays* se comprueben en cada acceso a los mismos. Si el compilador es capaz de averiguar si un índice usado para acceder a un elemento de un array va a estar dentro de los límites del mismo, entonces esa comprobación es redundante y se puede eliminar.
- **Asignación de registros por coloreado de grafos**: [Cooper05b] Una de las formas que existen de optimizar la ejecución de un programa es guardar los valores más usados por el código del mismo en las zonas de memoria más rápidas disponibles: los registros. Por ello, los compiladores suelen poseer una fase que se encarga de determinar qué variables deben guardar en los registros que estén disponibles en la arquitectura donde se ejecutan. Encontrar la solución óptima para determinar qué registros se deben guardar es un problema NP-completo, por lo

que normalmente se recurre a técnicas heurísticas que convierten el problema en un problema de coloreado de grafos (a partir de un grafo, se trataría de hallar el número mínimo de colores que hace que dos nodos unidos no puedan tener el mismo color). Los compiladores implementan esta técnica por tanto para tratar de hallar la mejor forma de disponer los valores que se usan en los registros del procesador, y de esta forma optimizar la ejecución del programa.

Teniendo en consideración todo lo explicado, *Sun Microsystems* proporciona en sus distribuciones de *Java* dos máquinas virtuales: La máquina cliente y la máquina servidor. La primera de ellas es más simple, y está pensada para una compilación rápida del programa efectuando pocas optimizaciones. La segunda produce código mucho más optimizado a cambio de un tiempo de compilación significativamente mayor [Doederlein03]. Cada usuario podrá escoger de esta forma la estrategia que más le convenga en cada uno de sus programas, seleccionando aquél que se adapte mejor a sus necesidades.

7.3 PANORÁMICA DE UTILIZACIÓN DE LA COMPILACIÓN JIT

Una vez definidas una serie de técnicas de optimización dinámica de código en la sección anterior, analizaremos en esta sección la técnica que probablemente se tome como base (puede y suele actuar conjuntamente con otras técnicas) para implementar sistemas con fines actualmente: la compilación *JIT*. Para ello se hará una panorámica de su utilización a lo largo de la historia en diferentes plataformas, contemplando cómo se ha implementado y los beneficios que se han pretendido lograr con la misma en cada una de ellas, de forma que se vea claramente la evolución de la misma y las diferentes formas de ser aplicada.

En esta sección pues presentaremos una serie de aplicaciones de la compilación *JIT* ordenadas principalmente por cronología, basándonos en el trabajo realizado por [Aycock03] al respecto. Algunas de estas aplicaciones emplean la compilación *JIT* de forma conjunta con otras técnicas de optimización, de forma que se pueda lograr un mejor rendimiento final.

La implementación de un compilador *JIT* suele estar fuertemente vinculada a un lenguaje concreto, de manera que esta técnica se suele asociar casi indivisiblemente con un lenguaje o plataforma en muchas ocasiones. Es por ello que esta sección se dividirá por lenguajes en vez de por otros criterios que pudiesen resultar más artificiales.

Otra de las consideraciones que haremos en esta sección es la de contemplar sólo implementaciones operativas de esta técnica de compilación, tratando de evitar en la medida de lo posible, salvo que sea razonable su inclusión, estudios preliminares o desarrollos que no tengan una implementación operativa asociada.

7.3.1 Lenguaje LC2

LC2, también denominado "lenguaje para la computación conversacional"

(*Language for Conversational Computing*), es un lenguaje diseñado para la programación interactiva [Mitchell68], de carácter principalmente experimental. Las técnicas empleadas por Mitchell en este lenguaje influyeron en el desarrollo de la compilación *JIT* en lenguajes como *Smalltalk* y *Self*, de manera que se puede considerar este lenguaje como un "pionero" en la inclusión de estas técnicas.

Mitchell implementó un sistema en el cual se podía generar código compilado a partir de un intérprete en tiempo de ejecución, guardando las acciones realizadas por el código durante la propia interpretación. Obviamente, esto sólo se puede hacer una vez el código ya ha sido ejecutado, pero es una mejora en la eficiencia de la ejecución de un programa interpretado, ya que la siguiente ocasión en que un mismo código vuelva a ser ejecutado, esta ejecución se hará a una velocidad mayor al ser código ya compilado y no tener que volver a interpretarse. Como puede verse, este lenguaje establece una base para lo que posteriormente se desarrollaría en la compilación *JIT* tal y como la conocemos actualmente.

7.3.2 Lenguaje APL

El lenguaje de programación *APL* (*A Programming Language*) es un lenguaje de programación orientado al trabajo con *arrays*, creado por Kenneth E. Iverson [Iverson62] con la intención de emplear una notación consistente para analizar diversos aspectos relacionados con las posibles aplicaciones de los computadores (y que de hecho fue usada para describir la arquitectura *IBM/360* [Amdhal00]). Este lenguaje es interpretado e interactivo, y las implementaciones más modernas incorporaban un soporte para programación modular y estructural. Aunque este lenguaje usa un conjunto de caracteres especial, una posterior evolución (el lenguaje de programación *J*) eliminó ese problema empleando *ASCII*. Abrams [Abrams70] fue el creador de una implementación eficiente de este lenguaje, describiendo dos estrategias de optimización:

- *Drag-along*: Esta técnica consiste en posponer la evaluación de una expresión hasta el último momento posible, con la intención de que en ese espacio de tiempo se pueda recabar toda la información de contexto necesaria para buscarle el método de evaluación más eficiente posible. Esta técnica es denominada actualmente "evaluación perezosa" (*lazy evaluation*).
- *Beating*: Consiste en una transformación del código para reducir la cantidad de datos que son manipulados a la hora de evaluar una determinada expresión.

Ambas técnicas descritas están íntimamente relacionadas con el proceso que se realiza en la compilación *JIT. APL* es un lenguaje dinámico, donde el tipo y los atributos de los objetos no son en general conocidos hasta el tiempo de ejecución. Para poder hacer optimizaciones realmente efectivas, éstas deben pues ser aplicadas cuando toda la información en tiempo de ejecución esté disponible.

Por otra parte, la "máquina *APL*" optimizada creada por Abrams empleaba dos compiladores *JIT* separados. El primero de ellos traducía los programas *APL* a un código en notación postfija para una máquina denominada "máquina D", encargada de simplificar el código *APL* aplicando las dos técnicas descritas anteriormente. Posteriormente, se invocaría una denominada "máquina E", que ejecutaría las instrucciones cuando fuese necesario. El objetivo de este sistema es lograr un soporte eficiente de *APL*, estableciendo una base para lograr el soporte *hardware* de este lenguaje de alto nivel.

7.3.3 Lenguaje BASIC

El lenguaje *BASIC* (*Beginner's All-purpose Symbolic Instruction Code*) [ANSIBASIC87] fue empleado como lenguaje de pruebas para una técnica denominada "*throw-away compilation*". En todo proceso de compilación *JIT* se debe hacer una balance entre velocidad de ejecución y espacio ocupado, teniendo en cuenta también que la mayoría de programas ejecutan una porción relativamente pequeña de su código la mayor parte del tiempo que están en ejecución [Knuth71]. La técnica de compilación mencionada es considerada como uno de los medios más adecuados para hacer un buen balance entre velocidad y espacio ocupado en memoria [Hammond77], mientras que el mencionado principio de localidad [Denning05] existente en los programas se tratará adecuadamente mediante otra técnica denominada "*mixed code*". Describiremos a continuación ambas:

- *Mixed code*: Esta técnica consiste en que el código de un programa resida en memoria como una mezcla entre código nativo y código interpretado [Dakin73] [Dawson73]. Las partes del código que se ejecutan más frecuentemente estarán convertidas a código nativo, mientras que las otras permanecerán en forma de código interpretado. De esta forma se pretende mejorar la eficiencia y minimizar el espacio ocupado en memoria, ya que las partes más usadas del código estarán en la forma de ejecución más rápida y el código en su forma interpretada se espera que ocupe menos espacio que su equivalente nativo. Dentro de esta misma técnica, también se consideró la creación de un intérprete personalizado [Pitman87], que en vez de presentar un programa como una mezcla entre código nativo e interpretado convierta todo el código a instrucciones de una máquina virtual, compilándose enteramente el programa a este código.

  El mayor impedimento encontrado para la implementación de esta técnica era que exigía mantener en memoria un intérprete y un compilador al mismo tiempo, ocupando un espacio en memoria excesivo para el *hardware* de esa época, lo que motivó la aparición de la siguiente técnica.

- Compilación "*throw-away*": Esta técnica tiene como objetivo principal la optimización del espacio ocupado por un programa. Mediante la misma se sustituye un proceso tradicional de compilación estática por otro en el que partes de un programa pueden ser compiladas dinámicamente según sea necesario. Antes de agotar el espacio disponible en memoria, cualquiera de estas partes compiladas dinámicamente pueden ser eliminadas de memoria para dejar espacio a partes nuevas, debiendo recompilarse de nuevo si vuelven a ser necesarias. De esta forma, se optimiza el espacio ocupado por un programa asegurándose de que sólo aquel código que sea más frecuentemente usado resida en memoria en un momento dado.

7.3.4 Lenguaje Fortran

El lenguaje *Fortran* (*Formula Translating System*) [ANSIFORTRAN97] es un lenguaje de propósito general, procedural e imperativo que está orientado a la computación numérica y científica. Este lenguaje ha evolucionado incorporando características como módulos, objetos y programación genérica (*Fortran* 90 y *Fortran* 2003). También albergó algunos de los primeros estudios de sistemas *JIT* en los que los programas optimizaban sus "*Hot Spots*" (puntos del programa que más se ejecutan) [Hansen74]. Este estudio contemplaba tres aspectos principales:

- Localizar el código que debe ser optimizado: La localización de "*Hot spots*" se realizaba de una forma sencilla, mediante un contador que permita medir la frecuencia de la ejecución para cada bloque de código.
- Determinar cuando es adecuado optimizar código: El contador de frecuencia de ejecución sirve también para determinar cuando su bloque de código asociado puede pasar a un "nivel" de optimización superior. Para ello, en un periodo intermedio entre la ejecución de diferentes bloques de código se invoca otro código de supervisión de bloques que trate los contadores de la forma adecuada, haga las optimizaciones necesarias y transfiera la ejecución al siguiente bloque.
- Determinar cómo se va a optimizar el código: Para la optimización del código se emplean una serie de estrategias de optimización convencionales, dependientes e independientes de la máquina, determinando el orden más adecuado para las mismas, de manera que un mismo bloque pueda ser optimizado por técnicas diferentes en momentos distintos. El compilador pues añadirá las optimizaciones de manera incremental a un mismo código, consiguiendo de esta forma limitar el tiempo empleado en las mismas (ya que en cada fase se aplicará un número limitado de éstas, en lugar de todas las posibles a la vez) y haciendo que el código más usado también sea el más optimizado (cuantas más fases de optimización se apliquen sobre el código, más rendimiento tendrá, al menos teóricamente).

Existen adicionalmente otras implementaciones de *Fortran* [Ng76] que compilan su código a seudo-instrucciones que luego serán interpretadas dinámicamente en tiempo de ejecución.

7.3.5 Lenguaje Smalltalk

El lenguaje *Smalltalk*, ya descrito en una sección anterior, es también un vehículo usado para optimizaciones basadas en compilación *JIT*. En este caso las optimizaciones se basan en seleccionar la representación de la información más eficiente posible de forma transparente al usuario [Deutsch84]. Determinadas implementaciones del lenguaje también usa una compilación *JIT* "estándar", donde las funciones se compilan a código nativo de forma perezosa en el instante en el que van a ser ejecutadas, y luego son introducidas en una zona de memoria caché para futuros usos. También se contempla un procedimiento de compilación "*throw-away*" ya descrito anteriormente.

7.3.6 Lenguaje Self

El lenguaje *Self*, también descrito anteriormente, es un lenguaje que, debido a ciertas características de su diseño (sistema de tipos dinámico, basado en prototipos,...) puede resultar un poco más difícil de implementar eficientemente. Por ello, posee un avanzado sistema de compilación *JIT*, consistente en la creación de tres generaciones distintas de compiladores [Hölzle94b], todos ellos invocados dinámicamente en el momento en el que se llama a un método, como en el caso de *Smalltalk*:

- Primera generación: Casi todas las técnicas de optimización empleadas por este

lenguaje se basaban en la información acerca de los tipos de los identificadores de un programa, y la transformación del mismo para que se pudiese tener alguna información fiable respecto a ellos, siendo sólo unas pocas las que estaban directamente relacionadas con la compilación *JIT*. En este compilador de primera generación, una de estas técnicas es la denominada "personalización" (*customization*), por la que en lugar de compilar dinámicamente un método a código nativo que pueda ser ejecutado en todas las llamadas siguientes al mismo, el compilador producirá una versión del método adaptada a un contexto particular, apoyándose en que, en tiempo de ejecución, el *JIT* dispondrá de mucha más información que cualquier proceso de compilación estática del mismo código, usando pues dicha información para generar un código mucho más eficiente. Las llamadas al método desde contextos similares podrán compartir este mismo código adaptado, pero también puede acarrear problemas si la adaptación del código se sobreutiliza, por la memoria que pueden ocupar todos los elementos que se generan en el proceso.

- Segunda generación: El compilador de *Self* de segunda generación amplia las técnicas de transformación usadas por su predecesor actuando en este caso sobre los bucles de un programa [Chambers90], siempre usando la información disponible en tiempo de ejecución. El código generado por este compilador es considerablemente más eficiente que el producido por la anterior generación. El proceso de compilación se agiliza no generando código para ciertas operaciones poco comunes que pueden ocurrir en el programa, que sólo se creará en tiempo de ejecución si estos casos se manifiestan [Chambers91].
- Tercera generación: La tercera generación del compilador *Self* trata de minimizar los tiempos en los que un programa está detenido por procesos de compilación en tiempo de ejecución, agrupando en la medida de lo posible las pausas producidas por estos eventos para que el usuario no perciba interrupciones constantes durante la ejecución de un programa dado [Hölzle94] [Hölzle94b]. De esta forma, se emplea un proceso de optimización adaptativa similar al visto en *Fortran*, usando un compilador rápido, que no efectúa procesos de optimización, para una compilación inicial de un método y manteniendo contadores de frecuencia de invocación que determinarán cuando una recompilación para optimizar el código debería ocurrir. De todas formas, el número de veces que un bloque es ejecutado no es el único criterio usado para determinarlo. La optimización *JIT* del compilador de *Self* emplea también el denominado "*feedback* de tipos" [Hölzle94b], mediante el cual, mientras un programa está en ejecución, la información de sus tipos es recogida por el sistema automáticamente y cuando ocurra una recompilación se pueda recurrir a dicha información para permitir una optimización más agresiva.

7.3.7 Lenguaje Oberon

Oberon es un lenguaje creado por Niclaus Wirth y otros investigadores del *ETH* de Zurich [ETHOberon06], muy similar a *Modula-2* (del mismo autor) pero con una extensión que le da capacidades de orientación a objetos. Este lenguaje fue usado como base para estudiar un sistema relacionado con la compilación *JIT* para la optimización de programas en tiempo de ejecución, que también aportaba una solución para el problema de la distribución de *software* en diferentes plataformas, denominado "*slim binaries*".

Tradicionalmente la distribución de un mismo programa en un entorno de computación heterogéneo es problemática, dadas las diferentes arquitecturas que pueden existir en dicho entorno que exigirían varios ejecutables adaptados a cada una de ellas. Dentro de una misma línea de procesadores inicialmente compatibles a nivel de código nativo, incluso es necesaria la distribución de múltiples ejecutables para poder

aprovecharse de las características concretas de cualquiera de ellos, adaptando el código para que se ejecute de la forma lo más eficiente posible. Para solucionar este problema se creó el concepto de "*slim binaries*" [Franz94].

Un "*slim binary*" contiene una representación de alto nivel e independiente de la máquina de un módulo de un programa. Cuando este módulo es cargado en memoria, el código ejecutable es generado dinámicamente, adaptándose a las características concretas del entorno donde se ejecuta. Nótese que en este caso el código se genera para todo el módulo en lugar de por cada procedimiento, algo que el autor consideró más eficiente. Para que este proceso fuese lo más efectivo posible, se necesita una generación de código rápida, para lo cual las estructuras de datos fueron optimizadas y el código ya generado se guarda para no tener que regenerarlo si se vuelve a necesitar.

Este sistema fue implementado sobre el lenguaje *Oberon*, y a partir del mismo se investigaron otras técnicas, como la optimización continua en tiempo de ejecución [Kistler99], donde partes de un programa en ejecución podrían someterse a un proceso de optimizaciones continuo sin un punto de parada definido, siempre y cuando pueda obtenerse una solución más óptima que la anterior. La optimización continua está orientada a aquellos contextos en los que los datos tomados para realizar una técnica de optimización concreta pueden variar (por ejemplo, ajustar una serie de datos para adaptarlos al patrón de acceso a los mismos que vaya a usar un programa concreto). El proceso de optimización continua se ejecuta en segundo plano, como un hilo de baja prioridad que trabaja durante el tiempo en el que el programa principal está parado.

Una idea similar a la presentada se ha implementado también sobre el lenguaje *Scheme* [Burger97], donde los bloques de código se reordenaban usando información de un *profiler* para mejorar la predicción de saltos *hardware*, entre otras posibles optimizaciones.

7.3.8 Plantillas, ML y C

Incluimos estos dos lenguajes bajo un mismo epígrafe ya que existen soluciones relacionadas con la compilación dinámica similares en ambos. Esta solución se denomina compilación por fases o etapas (*staged compilation*), y consiste en dividir la compilación de un programa en dos etapas: Compilación estática y compilación dinámica. Antes de la ejecución, un compilador estático procesa una serie de "plantillas", construyendo bloques que son explorados en tiempo de ejecución por el compilador dinámico para, entre otras cosas, situar valores en los lugares dejados para este propósito en las mismas.

Las plantillas son especificadas por anotaciones del usuario, aunque existen propuestas para generarlas automáticamente [Mock99]. Aunque este proceso, tal y como está descrito, no parece tener mucha relación con la compilación *JIT*, las plantillas pueden estar codificadas de forma que se requiera una traducción en tiempo de ejecución antes de ser ejecutadas, pudiendo pues el compilador dinámico hacer optimizaciones a las mismas antes de que éstas sean procesadas. Este método ha sido empleado en el lenguaje *ML* [Leone94], para la especialización en tiempo de ejecución en *C* [Consel96] y también para crear extensiones dinámicas de dicho lenguaje [Auslander96]. El sistema *Dynamo* [Dybvig06] propone de hecho hacer compilación por etapas y optimización dinámica para *Scheme*, *Java* y *ML*.

7.3.9 Erlang

Erlang es un lenguaje funcional diseñado para su empleo en grandes sistemas en tiempo real [Armstrong97]. Existe una implementación de un compilador *JIT* para este lenguaje llamado *HiPE* (*High Performance Erlang*) [Johanson00], que trata de solventar sus problemas de rendimiento y que permite la implementación eficiente de sistemas concurrentes de programación que usen paso de mensajes. En concreto, el sistema trata de investigar técnicas que reduzcan la necesidad de leer y guardar información de referencias en memoria, así como las diferentes ramas de ejecución de un programa para lograr programas concurrentes lo más secuenciales posible [HiPE06]. Este compilador necesita que el usuario lo invoque explícitamente, proporcionándole un alto grado de control sobre el balance entre el espacio ocupado por el código y su rendimiento.

7.3.10 O'Caml y Especialización

O'Caml es otro lenguaje funcional, dialecto del lenguaje *ML* [Rémy99]. El intérprete de este lenguaje tiene como objetivo la especialización de un programa en tiempo de ejecución. Existen trabajos de especialización de las instrucciones de un intérprete de una forma limitada [Piumarta98]. Para ello, se traducen los *bytecodes* que deberían ser interpretados en código que se ejecutará directamente en hilos de ejecución, combinando, en una segunda fase, determinados bloques de instrucciones en "macro *opcodes*", y modificando el programa para que use estas nuevas instrucciones. La intención de esta optimización es la de disminuir el coste introducido por el procesamiento de instrucciones, así como la de proporcionar nuevas oportunidades para realizar optimizaciones específicas con estos "macro *opcodes*", que no hubieran sido posibles con el conjunto de instrucciones original. No obstante, esta técnica no toma en consideración los procesos que tienen lugar durante la ejecución del programa para la optimización y está más orientada a conjuntos de instrucciones de bajo nivel.

Una aproximación más general a la especialización de programas fue la llevada a cabo mediante el sistema *Tempo* [Thibault00] [Consel98], aplicado a la máquina virtual de *Java* y un intérprete de *O'Caml* en tiempo de ejecución, aunque los resultados obtenidos por esta técnica no han sido tan efectivos como los proporcionados por otras.

7.3.11 Prolog

Prolog también posee un sistema de compilación dinámica, aunque el modelo de ejecución de *Prolog* hace necesario usar técnicas especializadas. Distintos trabajos acerca del tema [VanRoy94] [Haygood94] describen un proceso por el cual el código generado por un compilador nativo de *Prolog* puede ser cargado dinámicamente.

7.3.12 Simulación, Traducción Binaria y Código Máquina

La simulación es el proceso por el cuál se ejecutan programas nativos de una arquitectura A sobre otra arquitectura B. Una de las técnicas empleadas para la simulación está precisamente derivada de la compilación *JIT*, y consiste en la traducción binaria dinámica (*dynamic binary translation*), que implica la transformación de un código en otro en tiempo de ejecución. Los traductores binarios son programas altamente especializados, al depender fuertemente de la máquina origen y la de destino para la que han sido creados. El trabajo en traductores que permitan varios orígenes y/o destinos aún es un campo que necesita más desarrollo [Ung00]. Los simuladores pueden ser catalogados en cuatro generaciones según su implementación [May87] [Aycock03].

- Primera generación: En esta generación los simuladores eran intérpretes que simplemente procesaban cada instrucción del código fuente según fuese necesario, proceso mediante el cual no puede lograrse una eficiencia elevada.
- Segunda generación: Estos simuladores traducen las instrucciones del código original a código destino una a una, pero guardan en una memoria caché las transformaciones que realizan para usarlas posteriormente, aumentando de esta forma la eficiencia del proceso.
- Tercera generación: Para mejorar el rendimiento respecto a la generación anterior se traducen dinámicamente bloques de instrucciones completos de cada vez. Muchos de estos sistemas traducen bloques básicos o extendidos de código [Cmelik1994] que reflejen el flujo de ejecución estático del código fuente.
- Cuarta generación: Estos simuladores son los que predominan actualmente y sobre los que se desarrollan las últimas investigaciones [Bala99] [Chen2000]. Se mejora el rendimiento ofrecido por los de tercera generación traduciendo dinámicamente caminos o trazas. Un camino refleja el flujo de ejecución del programa en tiempo de ejecución (no el flujo estático, que se analizaba en la generación anterior). Todos estos simuladores tienen una estructura similar:

    - Ejecución con *profiler*: El simulador debe identificar áreas de código que son frecuentemente utilizadas (*HotSpots*). Para determinar estas áreas, el *profiler* analiza el programa fuente para obtener toda la información que necesite, esperándose que el tiempo invertido en esta tarea sea inferior a la ganancia de rendimiento que se pueda obtener. Cuando las arquitecturas origen y destino son la misma o muy similares, el programa se ejecuta directamente por la *CPU* y el simulador obtiene el control periódicamente, bien modificando (instrumentalizando) el programa fuente [Chen2000] o por mecanismos menos intrusivos, como interrupciones.
    - Detección de caminos de ejecución frecuentemente usados: Estos caminos serán detectados mediante contadores que permitan determinar la frecuencia de ejecución de un determinado código [Zheng2000], aunque también se pueden emplear técnicas para identificar zonas de código con una alta probabilidad de ser usadas muy frecuentemente [Bala99]. Algunos caminos pueden ser estratégicamente excluidos si son demasiado complejos de analizar o traducir, siendo muy importante determinar cuales son los puntos más óptimos para iniciar y finalizar el análisis de un camino.
    - Generación de código y optimización: Una vez identificado un camino de ejecución frecuentemente usado susceptible de ser tratado, el simulador

traducirá el código correspondiente a código para la arquitectura destino o bien lo optimizará.

- Mecanismo de "escape" (*bail-out*): A la hora de hacer una optimización dinámica es posible que el rendimiento del programa se vea resentido por ella, obteniendo un resultado contrario al buscado. Por ello, se necesita un mecanismo que heurísticamente detecte estos problemas y pueda dejar el programa en su estado anterior [Bala99], evitando también casos demasiado complejos.

Otro tema que aparece en los trabajos recientes acerca de la traducción binaria dinámica es el soporte *hardware* específico para esta tarea, especialmente para traducir código de arquitecturas heredadas en código de arquitecturas *VLIW* (*Very Large Instruction Word*) que tienen un mejor rendimiento y un paralelismo superior a nivel de instrucciones, mayores frecuencias de reloj y mejor consumo energético [Ebcioglu96] [Altman00] [Klaiber00].

7.3.13 Java

La plataforma *Java*, como ya se ha mencionado, compila el código del lenguaje a *bytecodes* de la máquina *JVM* (*Java Virtual Machine*). En sus comienzos el proceso de traducción se basaba en un único intérprete para todo el proceso, con una penalización de rendimiento importante. Para mejorar este rendimiento se empleó la compilación *JIT* de los *bytecodes* generados, siendo este sistema el que llevó al resurgimiento de esta técnica en la actualidad, convirtiéndose en la más popular para los fines que persigue y también en una nueva fuente de investigaciones. Actualmente, las implementaciones de técnicas *JIT* para estas plataformas no se limitan simplemente a la traducción de *bytecodes* a código nativo directamente, sino que se efectúa una optimización del código al mismo tiempo, empleando algoritmos de optimización que permiten un balance adecuado entre la velocidad del código resultante y el coste del algoritmo de optimización aplicado [Burke99] [Bik99] [Krall97]. Esto se consigue combinando la técnica *JIT* con otras técnicas de optimización adaptativa, según lo visto en la sección anterior [SunHotSpot06].

Existen también otros estudios sobre la compilación *JIT* para *Java* además del mencionado, como estrategias basadas solamente en compiladores (sin la existencia de la acción de intérpretes) [Burke99] o la traducción de código *JVM* a código *Self*, para beneficiarse de las optimizaciones existentes en el compilador de este lenguaje [Agesen97] que se mencionaron anteriormente.

Otra técnica relacionada son las anotaciones [Azevedo99], que permiten disminuir el esfuerzo realizado para tareas de optimización antes de la ejecución de un programa, mediante la precomputación y marcado de los *bytecodes* con aquellas informaciones necesarias que puedan ayudar al *JIT* a hacer una optimización más eficiente.

Por último, también existen implementaciones de procesos de compilación continua para *Java*, aunque esto se excede ya el ámbito de la compilación *JIT* que estamos describiendo en esta sección [Plezbert97].

7.3.14 Plataforma .NET

La plataforma *.NET* es una de las últimas plataformas en incorporar la compilación *JIT* como un medio para acelerar la ejecución de sus aplicaciones, extendiendo aún más su popularidad gracias a que esta plataforma también goza de una elevada extensión. Este sistema sigue un proceso de compilación *JIT* típico similar a los ya descritos [Anthony05]. Un método es compilado siempre antes de su primer uso, y todos esos métodos compilados son guardados en una caché para su reutilización futura, variando el tamaño de esta caché en función del comportamiento del programa. Una vez que el número de métodos acumulados alcanza el tamaño máximo, el sistema puede elegir entre aumentar el tamaño disponible o bien liberar todos los métodos que no están en el ámbito actual, proceso conocido como *code pitching* [Stutz03]. Este mecanismo tiene dos posibles puntos en los que la pérdida de eficiencia puede ser problemática: recorrer todos los métodos compilados y recompilar todos los métodos que hayan sido eliminados de memoria por el mencionado proceso de *pitching*. No obstante, de esta forma el sistema consigue mantener un balance entre espacio ocupado y velocidad de ejecución. Los pasos concretos seguidos por el proceso de compilación *JIT* son [Notario06]:

- Importación: En esta fase el código *CIL* se transforma en una representación intermedia del *JIT*, efectuándose una verificación del código si es requerida. Para esta fase es necesario que el *JIT* se comunique intensamente con la máquina virtual para acceder a todas las estructuras necesarias.
- Transformación: En esta fase el compilador transforma las estructuras internas existentes para simplificar y optimizar el código, detectando funciones especiales que se puedan transformar en representaciones lo más óptimas posible y otro tipo de optimizaciones estándar.
- Análisis del grafo de flujo: Se efectúa este análisis "tradicional" para determinar el tiempo de vida de las variables, detección de bucles y otros datos usados en las siguientes etapas.
- Optimización: Aquí ocurren todas las optimizaciones cuyo coste es elevado.
- Uso de registros: Esta fase trata de elegir el lugar más adecuado para cada variable, basándose en el número de usos de la misma, teniendo en cuenta que una variable contenida en un registro tendrá un acceso mucho más óptimo pero que se cuenta con un número de registros limitado para este tipo de operaciones.
- Generación de código: Fase en la cual se genera código para la plataforma nativa de destino (*x86*), teniendo en cuenta qué procesador se usa para generar el código más óptimo para el mismo. También se guarda en esta fase información para el recolector de basura y el depurador.
- Emisión: En la última fase toda la información generada se une y se devuelve a la máquina virtual, que redirigirá el control al nuevo código generado.

CLR emplea dos compiladores *JIT* para la generación de código, así como una estrategia de compilación independiente no relacionada que puede o no usarse para un código concreto [Manzoor06]:

- Econo-*JIT*: Este compilador permite compilar programas muy rápidamente pero no optimiza el código que genera, permitiendo procesar el programa rápidamente a costa de una ejecución de su código más lenta. La aplicación más típica de este compilador es la ejecución de *scripts*.

- *JIT* estándar: Este compilador procesa el código más lentamente, pero sí optimiza el código generado, consiguiendo mejor rendimiento en tiempo de ejecución. Éste es el que el *CLR* usará para ejecutar la mayoría de código.
- Compilación en tiempo de instalación: Esta técnica permite al *CLR* compilar la aplicación a código nativo cuando el programa se instala. La instalación del programa será más lenta, pero estará más adaptada a las características particulares de la máquina donde se va a ejecutar y su velocidad se verá teóricamente aumentada gracias a ello.

Por último, cabe destacar que diferentes implementaciones del estándar *CLI* (del que *CLR* es una implementación más) pueden emplear *JIT* con ligeras diferencias conceptuales o bien orientados a determinados propósitos, como ocurre con el *JIT* de *SSCLI*, orientado a permitir una modificación más sencilla del mismo que facilite su estudio [Stutz2003]. También existen proyectos de *JIT* que se adaptan al tipo de plataforma al que está destinada una distribución concreta del entorno, como por ejemplo para la plataforma *Windows CE* [Pratschner2006] o *IA-64* [Kumar06].

7.4 CONCLUSIONES

Vemos pues cómo la compilación *JIT* es una técnica muy usada actualmente en diferentes sistemas para la mejora de la eficiencia de aplicaciones en tiempo de ejecución, existiendo un gran abanico de estudios e investigaciones relacionados con dicha técnica, orientados a la optimización de programas en diferentes contextos y para determinadas necesidades. Dada la vinculación de esta técnica con las máquinas virtuales, que como hemos visto son usadas para la implementación de muchos lenguajes dinámicos, es viable la exploración de una aplicación de esta técnica sobre alguna máquina virtual existente para la mejora de la eficiencia de lenguajes dinámicos.

Tal y como hemos visto, existen múltiples aplicaciones de esta técnica de compilación, y algunas de ellas enfocan el proceso de compilación a obtener el mayor rendimiento posible en determinados casos o aplicaciones concretas. En cambio, la aplicación de *JIT* sobre las máquinas virtuales de *Java* o *.NET* no adapta la optimización en tiempo de ejecución que realizan a ningún escenario particular.

Por último, cabe destacar cómo en la evolución histórica que se ha presentado en esta sección se ha visto cómo los diferentes conceptos y técnicas que están involucrados en un proceso de compilación *JIT* moderno han ido surgiendo paulatinamente, respondiendo a necesidades de desarrollo o en un afán de mejorar el rendimiento del código, que ha impulsado a la creación de estos mecanismos. La unión apropiada de todos ellos o su adaptación a determinadas características han originado las técnicas de compilación *JIT* que conocemos hoy en día.

SECCIÓN C: CREACIÓN DEL SISTEMA

8 ARQUITECTURA DEL SISTEMA

En este capítulo seleccionaremos de manera justificada los diferentes elementos que van a constituir el sistema finalmente construido, en función de lo visto en los capítulos anteriores acerca de los mismos. Solamente se enumerarán muy someramente los objetivos que se han pretendido alcanzar con la selección de cada uno de los elementos que formarán parte del sistema, ampliando su descripción en capítulos posteriores.

8.1 MÁQUINA ABSTRACTA

A partir del estudio realizado en el capítulo correspondiente a máquinas abstractas, se puede afirmar que una máquina abstracta es una plataforma que proporciona un gran número de ventajas, como portabilidad, independencia de la plataforma e interoperabilidad entre otras ya mencionadas, lo que ha impulsado su uso actualmente como plataforma para múltiples lenguajes de programación, pudiendo encontrar implementaciones tanto de lenguajes estáticos (*Java*, *C#*) como de lenguajes dinámicos (*Python*, *Perl*, *Smalltalk*) sobre una de estas máquinas. Aunque ya se han comentado ampliamente los beneficios que reporta el uso de una máquina abstracta en un capítulo anterior, los resumiremos orientándolos solamente a su aplicación a la implementación de lenguajes:

- **Procesamiento de lenguajes**: El empleo de una máquina abstracta permite que un lenguaje de programación concreto pueda ser implementado para diferentes arquitecturas de una forma más sencilla. La existencia de un código intermedio común a todos los lenguajes de la máquina permite que, al compilarse todo lenguaje a dicho código intermedio, sólo haga falta un traductor del código intermedio al código nativo de una arquitectura X para poder ejecutar cualquier lenguaje sobre la misma. De esta forma, todo lenguaje capaz de ser traducido a código intermedio será directamente soportado en la máquina X, realizándose así menos procesos de traducción (y herramientas los implementen).
- **Entornos de Programación Multilenguaje**: Las máquinas abstractas son unas herramientas adecuadas para diseñar entornos de programación que soporten múltiples lenguajes, ya que la compilación de todo lenguaje a un código intermedio común facilita que los lenguajes puedan interoperar entre sí. Si todo dato u operación de un lenguaje va a ser convertido a una representación intermedia común, entonces desde cualquier lenguaje se podría teóricamente acceder a dichos elementos aunque estén definidos en otros lenguajes de la máquina, ya que su representación siempre será idéntica, independientemente del lenguaje que se use para crearlos en primera instancia.
- **Portabilidad del Código**: El uso de una máquina abstracta permite la ejecución de cualquier programa en múltiples plataformas sin necesidad de modificarlo. Si un programa, realizado en un lenguaje soportado por la máquina abstracta, es compilado a un código intermedio propio de dicha máquina en lugar de a código

nativo de una arquitectura cualquiera, entonces sólo hará falta poseer una implementación de dicha máquina abstracta sobre otra arquitectura para que el programa pueda ejecutarse directamente sobre ella, sin necesidad de modificar en ningún caso el propio programa. De esta forma se pueden ejecutar programas idénticos sobre plataformas distintas.

- **Creación de un entorno computacional completamente orientado a objetos**: Una máquina abstracta permitiría la creación de un sistema computacional en el que sus programas usen el mismo paradigma que la propia máquina (por ejemplo la orientación a objetos), de manera que la funcionalidad desarrollada en estos programas pudiese formar parte de la propia máquina cuando se ejecuten, ampliando por tanto la funcionalidad del sistema de forma transparente al usuario.
- **Distribución e Interoperabilidad de Aplicaciones**: Gracias a que una máquina abstracta permite establecer una base sobre la cuál se puedan ejecutar múltiples lenguajes que finalmente serán traducidos a una forma intermedia común, es mucho más sencillo lograr la cooperación entre programas diseñados por distintos lenguajes, bien por permitir usar módulos de uno desde el otro (interoperabilidad) o bien por permitir dividir sus funcionalidades de manera que se puedan ejecutar en múltiples máquinas situadas en puntos geográficos distintos (distribución).

Debido a todas estas ventajas, se ha invertido un considerable esfuerzo de investigación en la mejora de su rendimiento, mediante las técnicas de optimización que también han sido estudiadas en esta tesis. Todo lo expuesto hace que se haya tomado la decisión de que la base sobre la que el sistema se va a construir sea precisamente una máquina abstracta, que permitirá al sistema final gozar de todas estas ventajas y aprovecharse de los avances en el área de rendimiento que han sido creados para ellas, con la intención de que con esa combinación de elementos se logre un mejor rendimiento que otras implementaciones que también proporcionen al usuario las mismas primitivas de reflexión. Una vez tomada esta decisión primordial, ante el amplio conjunto de máquinas existentes y potencialmente seleccionables para nuestros fines, se debe tomar una decisión adicional: Seleccionar aquélla que nos sea más apropiada para la consecución de nuestros requisitos. Para ello, de entre las máquinas estudiadas en el estado del arte, y de acuerdo con los requisitos expresados al comienzo de esta tesis, debemos seleccionar aquélla que mejor se adapte a estas condiciones:

- Servir para realizar programas de propósito general.
- Que pueda ser usada, directa o indirectamente, para el desarrollo de aplicaciones comerciales.
- Que posea mecanismos de optimización dinámica de aplicaciones en tiempo de ejecución.
- Que tenga un rendimiento base aceptable.
- Que permita su modificación interna con el mayor grado de libertad posible y que su código base tenga una estabilidad (entendiendo por ello que el sistema no presente errores evidentes o actualizaciones periódicas constantes en su código).
- Que soporte múltiples lenguajes y puedan crearse nuevos lenguajes sin muchas dificultades.
- Que permita interoperabilidad entre sus distintos lenguajes soportados.

Con este conjunto de características, la selección de la máquina abstracta más adecuada de todas las vistas es más sencilla. La necesidad de un sistema de propósito

general descarta inmediatamente cualquier sistema de propósito específico (como portabilidad de código, interoperabilidad entre aplicaciones, protección de la propiedad intelectual,...). Por otra parte, la necesidad de que la máquina virtual seleccionada tenga una vocación de desarrollo de *software* de carácter comercial descarta también la selección de sistemas que estén en fase de prototipo, experimentales o que no permitiesen (o proporcionase soporte para) la creación de aplicaciones comerciales según las tendencias existentes actualmente.

Por todo lo dicho anteriormente, el abanico de sistemas potencialmente seleccionables se ve bastante reducido, facilitándonos la selección. Para afinar aún más la misma, podemos decir que, sin la presencia de las dos últimas condiciones, solamente las máquinas virtuales relacionadas con la plataforma *Java* y el estándar *CLI* serían, a nuestro juicio, las que pueden cumplirlas de forma más conveniente. Sin embargo, estas dos importantes condiciones finales hacen que la plataforma seleccionada finalmente sea una basada en el estándar *CLI*, ya que si bien ambas plataformas permiten la creación de nuevos lenguajes destinados a ejecutarse sobre su máquina virtual (aunque en el *CLI* esta posibilidad ha sido más desarrollada, existe un mayor número de lenguajes de diferente carácter destinados a la misma), sólo en el *CLI* se da la condición de interoperabilidad de la forma establecida en los requisitos de esta tesis.

Nótese que en ningún caso se ha hecho mención al carácter estático de las máquinas virtuales mencionadas (en el sentido de que están orientadas inicialmente al soporte de lenguajes estáticos y no a los dinámicos), ya que los problemas derivados de este aspecto serán parte de los que tendrá que resolver la modificación que efectuemos a la máquina seleccionada. Es precisamente el no haber encontrado un sistema basado en máquinas abstractas que soporte nativamente lenguajes estáticos y dinámicos, posea técnicas de optimización en tiempo de ejecución para sus programas, de carácter comercial y completamente desarrollado, lo que motiva el desarrollo de esta tesis.

Una vez seleccionado un sistema basado en el estándar *CLI* para el desarrollo, quedaría estudiar y seleccionar la implementación concreta del mismo más adecuada para nuestros propósitos, algo que se hará en un capítulo posterior. El diseño de las modificaciones se hará por tanto, sobre la base de los elementos que define el propio estándar, personalizándolo para adaptarlo a los detalles de la implementación concreta en el mismo capítulo anterior.

8.2 OPTIMIZACIÓN DINÁMICA DE APLICACIONES

El mecanismo de optimización de aplicaciones dinámico usado en la arquitectura del sistema a construir estará fuertemente condicionado por la máquina abstracta escogida, ya que normalmente sólo poseerá una clase principal de mecanismo de este tipo. En este caso, el método de optimización dinámica usado por todas las implementaciones de *CLI* consideradas es la compilación *JIT*, y por tanto éste será el elegido. Por otra parte, este método de optimización, junto con otras técnicas adicionales, es uno de los más usados para este propósito hoy día (actualmente no es posible encontrar un sistema comercial, de características equivalentes a las implementaciones del estándar *CLI,* que implemente de la misma forma algunas de las otras técnicas de optimización vistas) y consigue buenos resultados, por lo que su selección debería proporcionarnos un rendimiento adecuado. Además, otras técnicas de optimización vistas presentan ciertos aspectos que dificultan su implementación o la hacen inadecuada para ciertas aplicaciones.

Como hemos visto, la compilación *JIT* permite transformar el código de la aplicación a código nativo en la propia máquina donde se llevará a cabo la ejecución dinámicamente, pudiendo así generarse código adaptado a las características concretas de la propia ejecución y del entorno donde el programa está siendo procesado en un momento dado, proporcionando teóricamente un mejor rendimiento. Esta técnica permite además un adecuado balance entre rendimiento obtenido y espacio en memoria ocupado, lo que la hace muy adecuada para los fines de esta tesis, donde no se busca rendimiento a cualquier precio, sino asociado a un coste de memoria razonable.

Algunas de las implementaciones consideradas pueden combinar la compilación *JIT* con otras técnicas de optimización adicionales, pero esto dependerá de la plataforma final seleccionada. En cualquier caso, se procurará hacer uso de todas aquellas técnicas de optimización que la implementación del *CLI* finalmente seleccionada pueda aportar.

Por otra parte, cabe destacar como hoy en día existen implementaciones plenamente operativas y completas de lenguajes dinámicos sobre máquinas virtuales que incorporan compilación *JIT*, como las implementaciones de *Python* sobre *.NET* [IronPython06] y la máquina virtual de *Java* [Jython06] o la implementación del lenguaje *Ruby* sobre una la virtual *.NET* [Gardens06]. Estas implementaciones logran una integración con la máquina virtual y pueden usar código desarrollado con otros lenguajes de la misma, al generar el mismo código intermedio que ellos. Además de esta forma se aprovechan de las ventajas que les da la máquina virtual sobre la que se implementan, ventajas que ya han sido descritas anteriormente.

No obstante, conviene recordar que estas máquinas están altamente optimizadas para la ejecución de lenguajes estáticos y por ello todas las características flexibles de estos lenguajes tendrán que ser emuladas (incorporando código adicional y por tanto una penalización en el rendimiento sobre las operaciones que requieran del uso de flexibilidad), ante la carencia de soporte directo dentro de la propia arquitectura del sistema para muchas de estas características. Esta tesis trata precisamente de superar estos problemas dotando a la máquina de partida de soporte nativo para las características flexibles de estos lenguajes, es decir, dotar a la máquina de soporte para un mayor nivel de reflexión adecuado a los lenguajes dinámicos (que como hemos visto en el estudio de los sistemas existentes, el nivel de reflexión usado por la gran mayoría de los lenguajes dinámicos más extendidos hoy en día es la reflexión estructural).

8.3 NIVEL DE REFLEXIÓN

Anteriormente hemos visto cómo la reflexión es un método adecuado para lograr la flexibilidad dinámica del sistema sin establecer necesariamente restricciones previas a aquello que puede ser reflejado. Dentro de la clasificación de los diferentes tipos y niveles de reflexión ya vista, usaremos aquélla que se refiere a lo que es reflejado para seleccionar el mejor conjunto de operaciones reflectivas a implementar. Como se ha visto, dentro de esta clasificación podemos encontrar tres niveles de flexibilidad:

- **Introspección**: El nivel más básico de reflexión, implementado por muchos sistemas. El estándar *CLI* ya contempla este nivel, que por otra parte no es suficiente para dar un soporte adecuado a lenguajes dinámicos.
- **Reflexión estructural**: Este nivel de reflexión es el poseído por muchos lenguajes dinámicos actuales y combina un alto nivel de flexibilidad con un coste de implementación razonable.

- **Reflexión computacional**: Este nivel de reflexión obtiene el grado de flexibilidad del sistema más elevado, pero con un coste de implementación que puede ser muy alto. Alterar la estructura de clases e instancias (reflexión estructural) es menos costoso que modificar la semántica de las diferentes primitivas del sistema (sobre todo si se hace sobre un sistema ya existente que no se ha diseñado con esta premisa en mente), algo que muchas veces lleva a establecer limitaciones previas acerca de lo que se puede y no se puede modificar (estableciendo por ejemplo los ya mencionados protocolos de metaobjetos o *MOPs,* que precisamente controlarían este aspecto). Por tanto, el coste de implementación de este nivel de reflexión puede llegar a ser desproporcionado si se le compara con los beneficios que puede ofrecer.

Una vez visto esto se debe hacer una selección del nivel más adecuado de reflexión que debemos intentar incorporar al sistema modificado. Descartado el primero por no ofrecer una funcionalidad suficiente para implementar el nivel de reflexión ofrecido por los lenguajes dinámicos, la elección recae entre la reflexión estructural y la computacional. Para seleccionar uno de ellos, debemos tener en cuenta las siguientes consideraciones:

- La mayoría de lenguajes dinámicos más usados hoy en día implementan soporte para reflexión estructural.
- La reflexión estructural ofrece un compromiso adecuado entre flexibilidad y rendimiento.
- El sistema existente escogido será más sencillo de modificar si se implementa la reflexión estructural. La sencillez no sólo afecta a la dificultad conceptual o de implementación de las modificaciones, sino que hace menos probable una pérdida excesiva de rendimiento.
- Es posible emular el funcionamiento de determinadas primitivas computacionales mediante el uso de primitivas estructurales (por ejemplo, la modificación de la semántica de invocación a métodos puede simularse por una funcionalidad que permita encapsular dichos métodos e implementar esta semántica modificada antes de llamarlos, encapsulación que se realizaría dinámicamente con reflexión estructural), lo que permitiría gozar de capacidades proporcionadas por la reflexión computacional incluso sin implementar realmente este nivel de flexibilidad.

Las razones expuestas hacen que finalmente la decisión sobre el tipo de reflexión a implementar en el sistema recaiga sobre la reflexión estructural, al establecer un compromiso adecuado entre flexibilidad y coste computacional. Se detallarán más aspectos acerca del nivel de reflexión implementado en un capítulo anterior. Por otra parte, la reflexión a implementar también será dinámica, procedural y de acceso interno, de acuerdo con la clasificación realizada en el estudio de los sistemas existentes.

8.4 MODELO COMPUTACIONAL

Como veremos más en detalle en un capítulo posterior, el modelo basado en clases (que es el que poseerá el sistema base seleccionado, ya que todas las implementaciones disponibles actualmente del *CLI* lo usan) no es en sí suficientemente

potente para poder soportar todas las primitivas que son necesarias para implementar un soporte adecuado para reflexión estructural sin que se rompa su coherencia. Por otra parte, también hemos visto cómo el modelo basado en prototipos resulta más adecuado para este tipo de operaciones. Además, la incorporación del modelo de prototipos también tiene otras ventajas que se han visto en el capítulo dedicado al mismo en el estudio de sistemas existentes:

- **Reducción semántica**: Suprimir el concepto de clase elimina la dependencia entre objetos - clases.
- **Inexistencia de pérdida de expresividad**: La utilización de prototipos no supone una pérdida de expresividad. Cualquier aplicación hecha con el modelo de clases puede ser traducida al modelo de prototipos.
- **Traducción intuitiva**: La traducción entre ambos modelos es muy sencilla.
- **Soporte de reflexión**: Como se ha dicho anteriormente, este modelo soporta mucho mejor las primitivas de reflexión estructural.
- **Interoperabilidad de lenguajes con los dos modelos**: Dado que cualquier modelo de clases puede ser traducido a un modelo de prototipos equivalente, es posible establecer vínculos entre lenguajes que sigan ambos modelos y permitir que programas hechos en modelos diferentes puedan interoperar.

No obstante, no podemos implementar directamente este modelo sobre el sistema extendido, ya que se debe mantener la compatibilidad del sistema extendido con el código heredado. Por ello, el modelo computacional del sistema final será una combinación de características de ambos, creando pues un modelo híbrido, incorporando al modelo basado en clases existente aquellos elementos del modelo de prototipos que sean necesarios para dotar al sistema de soporte para primitivas de reflexión estructural sin alterar sus capacidades ya existentes. Esto permitirá hacer operaciones reflectivas sin provocar incoherencias en el modelo y permitir la ejecución de código heredado de la misma forma que lo permitía el modelo original.

Por otra parte, una vez desarrolladas las modificaciones al sistema, éste será capaz de compilar un conjunto más amplio de lenguajes de alto nivel. El sistema extendido final podrá compilar tanto lenguajes estáticos como lenguajes dinámicos a un código *CIL* común, que será transformado a código nativo susceptible de ser ejecutado en la máquina destino. La sintaxis del *CIL* no será modificada para ello, y cualquier lenguaje podrá generar el mismo conjunto de *opcodes*, siendo la sintaxis de este *CIL* idéntica a la que ya poseía el sistema original. Además, todo lenguaje estático soportado por la máquina podrá tener acceso al conjunto de capacidades reflectivas implementado sin necesidad de ser modificado, mediante un mecanismo que le permita hacer un uso explícito de las mismas. El diseño del modelo computacional se realizará más adelante.

8.5 INTERACCIÓN DE MODELOS COMPUTACIONALES

Dado que, como hemos visto en el apartado anterior, se debe crear un modelo computacional híbrido nuevo para permitir la implementación de las primitivas dinámicas requeridas de forma coherente, se plantea entonces la situación de que dentro de la

misma máquina coexistirán dos modelos diferentes: El modelo Orientado a Objetos de clases tradicional que ya existe y el modelo nuevo creado para dar soporte a las nuevas características, basado en el de prototipos. En el sistema final, ambos modelos no deberán contemplarse como entidades separadas, y tendrán que ser capaces de interactuar entre ellos, de manera que cualquier programa desarrollado en uno de ellos pueda usar entidades y código desarrollados en el otro, permitiendo por tanto la interoperabilidad sin ninguna restricción. En concreto esta interacción entre lenguajes seguirá los siguientes principios:

- El compilador de cada lenguaje de alto nivel será el encargado de seleccionar el modelo computacional soportado.
- La máquina virtual soportará los distintos modelos por igual, de manera integrada en su arquitectura, y ofertará ambos a los lenguajes de alto nivel.
- Al estar dentro de la misma máquina, la interoperabilidad entre lenguajes deberá ser directa, es decir, los lenguajes mantendrán el mismo soporte para interoperabilidad existente actualmente entre todos los lenguajes destinados a la máquina virtual.

La siguiente figura muestra como se plantea estructurar la interacción descrita:

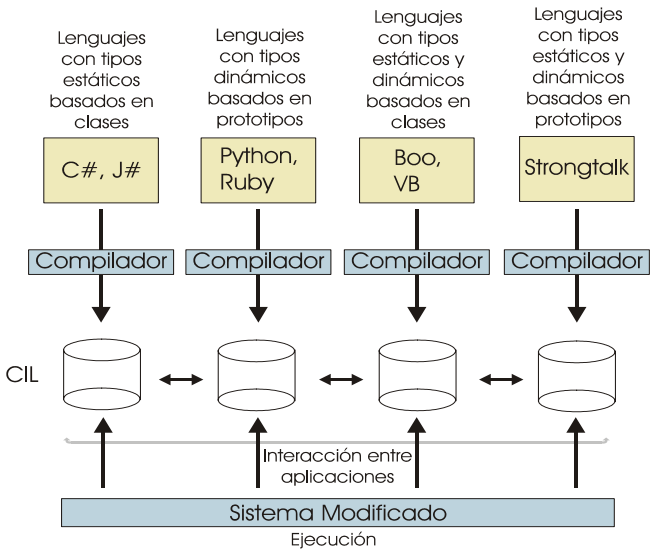

Figura 8.1. Interacción entre modelos computacionales

Posteriormente también se tratará de establecer el tipo de colaboración entre ambos modelos y el comportamiento esperado ante las diferentes opciones que se pueden plantear en estos casos.

9 MOTOR COMPUTACIONAL DEL SISTEMA

En este capítulo se describirán todos los elementos que formarán parte del motor computacional del sistema, lo que implica hacer una descripción de cómo se integrarán, a nivel de diseño, todas las modificaciones necesarias para cumplir con los objetivos planteados dentro del motor de la máquina virtual de partida, lo que determinará pues la configuración del motor computacional del sistema final. Una vez visto dónde se situarán los elementos necesarios, en los siguientes capítulos se hablará en profundidad cada uno de ellos. Ha de tenerse en cuenta que, si bien se ha seleccionado una plataforma de partida para hacer las modificaciones, no se ha determinado aún la implementación concreta de dicha plataforma que se va a emplear, por lo que tendremos que especificar el diseño de las modificaciones sobre entidades abstractas (hasta donde nos permita conocer el estándar, de forma que sólo se trabaje con elementos comunes a todas las implementaciones) y por ello el detalle de las mismas será limitado. Más adelante, cuando se seleccione la implementación concreta, podremos especificar más ciertas operaciones.

9.1 LA MÁQUINA ABSTRACTA

9.1.1 Uso de una Máquina Abstracta como Base del Sistema a Desarrollar

Como ya hemos mencionado, emplear una máquina virtual reporta una serie de beneficios importantes a la hora de desarrollar lenguajes sobre ellas, y usaremos por ello una máquina de estas características para desarrollar el motor computacional de nuestro sistema. No obstante, se debe tener en cuenta que el uso de una máquina virtual en principio plantea una desventaja frente a los sistemas nativos, ya que puede causar una pérdida de rendimiento más o menos significativa en la ejecución de los programas. Dado que un lenguaje de programación ya no es directamente traducido a código nativo ejecutable de la máquina destino, sino que pasa por varias etapas que conllevan un procesamiento adicional y más niveles computacionales intermedios, generalmente el código ejecutado sobre una máquina virtual es menos eficiente que aquél que se ejecuta directamente sobre la arquitectura nativa [Ballard95]. Sin embargo, hemos visto también que existen varias líneas de investigación y soluciones que mitigan las pérdidas de rendimiento, entre las que se encuentran las técnicas de optimización dinámica vistas en el capítulo correspondiente, que permiten el desarrollo de soluciones con una efectividad muy elevada, como en el caso de máquinas optimizadas para lenguajes estáticos (máquinas estáticas) como *JVM* o *CLR*, que gracias a ellos son usadas para desarrollar aplicaciones comerciales en diferentes ámbitos.

El problema del rendimiento debe tenerse en consideración especialmente en el caso de los lenguajes dinámicos. Debido a que estos lenguajes permiten al usuario un nivel de flexibilidad superior, el rendimiento en tiempo de ejecución de los programas creados con los mismos se ve afectado negativamente, al existir un mayor número de operaciones que el sistema necesitará hacer dinámicamente para soportar ese grado de flexibilidad. Hemos visto que existen múltiples ejemplos de lenguajes dinámicos que actualmente usan máquinas abstractas, creadas precisamente como soporte para la ejecución de sus programas, estando generalmente estas máquinas diseñadas específicamente para dicho lenguaje. Por tanto, una combinación de una máquina virtual y un lenguaje dinámico tendrá que ser especialmente cuidada en aspectos de optimización en tiempo de ejecución para que el rendimiento de los programas no se vea muy afectado.

En esta tesis se van a recoger todas las ideas mencionadas anteriormente, de manera que, dado que hoy en día las máquinas abstractas estáticas son las que más han avanzado en aspectos de optimización dinámica, tomaremos una de ellas con un buen rendimiento de base para elaborar una versión modificada que dé soporte nativo a las primitivas reflectivas que muchos lenguajes dinámicos usan para proporcionar flexibilidad a sus programas. El empleo de una máquina virtual existente, de proyección comercial y plenamente funcional permitirá que nos podamos aprovechar de todas sus características, mientras que la implementación dentro de la misma de primitivas reflectivas usará sus técnicas de optimización dinámica de programas para tratar de mejorar el rendimiento de las mismas. Esto permitirá estudiar si una plataforma de estas características es adecuada para mejorar el rendimiento de futuros lenguajes dinámicos que se ejecuten sobre ella usando dichas primitivas. Obviamente, para soportar primitivas reflectivas sobre una máquina estática habrá que hacer cambios en su modelo de computación, algo que veremos en el capítulo siguiente.

La figura 9.1 muestra a grandes rasgos cómo se ha planificado la integración los nuevos elementos a añadir en la arquitectura concreta de la máquina abstracta seleccionada (basada en el estándar *CLI*, como se vio en el capítulo anterior). Como se dijo al principio del capítulo, no se describirá aquí la arquitectura general de la máquina a modificar ya que no se ha seleccionado una implementación concreta aún (la descripción se hará en el apéndice A de esta tesis), por lo que sólo la mostraremos abreviadamente, con bloques que agrupen diferentes conceptos y subsistemas de forma general.

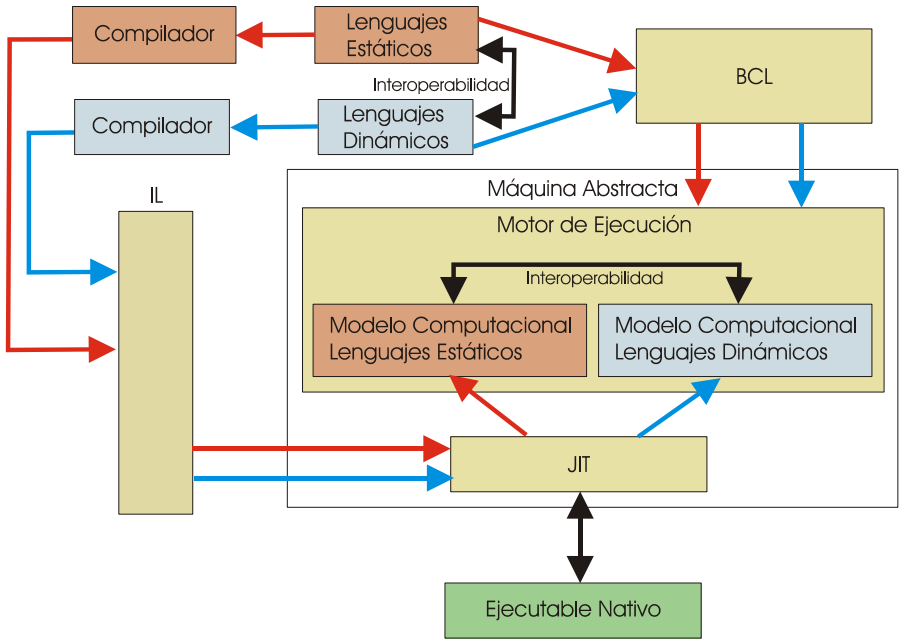

Figura 9.1: Diagrama de integración de nuevas capacidades en la *VM*

En la figura 9.1 se ve cómo la máquina extendida podrá soportar tanto lenguajes estáticos como dinámicos, pudiendo acceder ambos a todas las funcionalidades que le aporta la librería de clases del sistema (*BCL*) ya existente y permitiendo que ambos tipos de lenguajes interoperen. Cualquier lenguaje seguirá compilándose al mismo código intermedio *CIL* de la máquina original (necesario para permitir la interoperabilidad) que, gracias a la existencia de un compilador *JIT* modificado, podrá generar código ejecutable nativo empleando el modelo computacional más adecuado en cada momento (según se usen o no las capacidades de reflexión añadidas), permitiéndose así la convivencia de ambos tipos de lenguaje bajo la misma máquina. Las primitivas reflectivas a implementar se situarán en el propio motor de ejecución del sistema, al igual que el nuevo modelo computacional creado para soportarlas, para que así se pueda acceder a ellas tanto desde el código *CIL* como desde la librería *BCL*, a través de clases preparadas para ello, y que de esta forma cualquier lenguaje ya existente pueda hacer uso de las mismas. Vemos pues en este esquema general como el motor del sistema, su *JIT* y su modelo computacional deberán ser modificados en mayor o menor medida para llevar a cabo lo que se pretende hacer en esta tesis. En cualquier caso, debe quedar claro que la plataforma va a ser **extendida**, es decir, que en todo momento existirá una compatibilidad total con el código heredado.

9.1.2 Rendimiento y Compilación bajo Demanda

Una de las características implementadas por las máquinas abstractas para mejorar el rendimiento de las aplicaciones que se ejecutan sobre las mismas, aspecto capital en esta tesis como se ha destacado anteriormente, es la posibilidad que ofrecen estas máquinas de hacer compilación bajo demanda (*JIT*). A modo de ejemplo, citaremos a la máquina *.NET* de *Microsoft*, basada en el mismo estándar *CLI* que ha sido seleccionado como base para esta tesis. Cuando se compila una aplicación para *.NET*, ésta quedará traducida a código intermedio (*CIL*) en lugar de a código máquina como las aplicaciones "tradicionales" de *Windows* [Bühler06]. En el momento que la aplicación necesita ser ejecutada, el compilador bajo demanda de la infraestructura traducirá dicho código intermedio a código máquina usando diferentes técnicas y con una diferencia esencial respecto a la compilación tradicional (que se produce en el equipo del desarrollador), ya que la compilación bajo demanda se lleva acabo en el sitio donde finalmente residirá la ejecución. La principal ventaja que esta técnica posee es pues que el código puede ser optimizado de acuerdo a la configuración del equipo que se tenga en ese momento (cantidad de procesadores, el tamaño de su caché, el soporte para *HyperThreading*, etc.), adaptando pues el código a las características concretas de cada máquina. Por supuesto, este proceso pasa completamente desapercibido para el desarrollador.

Por tanto, la adaptación del código de forma transparente al usuario, para aprovechar mejor las características y configuración particular del sistema sobre el que se ejecuta, es una de las principales características que un sistema basado en máquina abstracta con compilación bajo demanda ofrece, y que aprovecharemos para el desarrollo del sistema propuesto en esta tesis. La compilación *JIT* es una forma de conseguir compilación bajo demanda y la usaremos para implementar las nuevas características de reflexión que incorporaremos de la forma que indicaremos en el siguiente apartado.

9.2 COMPILACIÓN JUST IN TIME (JIT)

Como ya se ha visto, el funcionamiento característico los sistemas de compilación bajo demanda hace que hoy en día constituyan el mejor compromiso entre rendimiento, portabilidad y también consumo de recursos. Los sistemas de compilación *Just In Time* (o *JIT*), como aplicación práctica de este tipo de sistemas, efectúan la traducción de código intermedio a código nativo en diferentes momentos de la ejecución del mismo, seleccionando las estrategias de traducción que mejor se adapten al programa concreto. Hemos visto por ejemplo que los sistemas *JIT* compilan los métodos sólo en el momento en el que son llamados por primera vez, traduciendo por tanto sólo aquéllos que realmente se usan durante una ejecución concreta de un programa, y que en muchas ocasiones emplean estrategias combinadas con otros sistemas de optimización dinámica para lograr los mejores resultados posibles. Además de las otras ventajas vistas, el uso de esta técnica como herramienta de optimización de lenguajes (dinámicos o estáticos) nos permitiría mover muchas de las operaciones más costosas de optimización de código a la fase de traducción a código intermedio, haciendo que la transformación de éste a código máquina dinámicamente sea más eficiente.

La aplicación de la compilación *JIT* sobre lenguajes dinámicos es posible y de hecho existen sistemas que se están creando con esa intención [Parrot06]. Un lenguaje dinámico que se compile mediante *JIT* se beneficiará de un tiempo de traducción menor (ya que las optimizaciones de coste elevado se habrán llevado a cabo al traducirlo a código intermedio), podrá gozar de un nivel de optimización variable y controlable (en función del tiempo que se quiera invertir en esa tarea en el proceso de traducción nativa) y ocupará menor espacio en memoria (ya que parte del programa estará en forma de código intermedio en tiempo de ejecución, forma en la que las instrucciones tendrán una mayor carga semántica y por tanto representarán la misma lógica en menos espacio).

Por último, no hay que olvidar que, dado que la compilación *JIT* se efectúa en plena ejecución del programa, se tendrá acceso a toda la información del mismo en tiempo de ejecución. Esto es algo muy importante de cara a incorporar soporte para lenguajes dinámicos, ya que este tipo de lenguajes necesitan hacer computaciones adicionales en ese momento debido a sus características. Por tanto, podremos "intervenir" en el proceso de traducción a código nativo efectuado por el compilador *JIT* de la máquina abstracta para implementar un soporte adecuado para las mencionadas capacidades reflectivas y en general para dar mejor soporte a lenguajes dinámicos.

En un proceso de compilación "tradicional" de un programa estático, una vez el programa es transformado a código nativo, normalmente la gran mayoría de metainformación del programa (estructura de los tipos, etc.) que es necesaria para la compilación es eliminada para hacer su ejecución más eficiente. Esto no ocurre sin embargo en una máquina abstracta como la usada para esta tesis, ya que realmente un fragmento de programa no se podrá ejecutar hasta que haya sido transformado en código nativo desde el código intermedio al que se compila y, dado que esa transformación ocurre dinámicamente, toda la información acerca de la estructura del programa estará disponible en ese momento para que dicha compilación pueda llevarse a cabo. Esto nos permitiría controlar cómo se efectúa esta transformación, de manera que a la hora de acceder a cualquier miembro de una clase u objeto se pueda consultar dicha información en busca de datos que hayan sido añadidos dinámicamente por el programa, actuando en consecuencia sobre el código generado. Por tanto, si modificamos dinámicamente la metainformación de un objeto o clase en un programa, usando un compilador *JIT* podremos intervenir, en tiempo de ejecución, el proceso de traducción para que éste genere código que dependa de la información que se haya añadido dinámicamente, ya que realmente dicho proceso tendrá que acceder a la misma para poder generar código nativo. Además, desde el propio código nativo generado también se deberá poder acceder a la información del programa, ya que durante la ejecución del

mismo también se puede requerir el acceso a la información añadida dinámicamente.

A modo de resumen, la aplicación de la compilación *JIT* para un lenguaje dinámico que se ejecute sobre una máquina virtual requiere dos condiciones generales:

- La existencia de un código intermedio sobre el que no se haga una comprobación de tipos estática al ser transformado a código nativo, ya que en ese caso no podremos adaptar correctamente la máquina. La traducción del lenguaje de alto nivel a código intermedio hará una serie de comprobaciones de tipo determinadas (que dependerán exclusivamente del lenguaje de alto nivel) y posteriormente el código intermedio será procesado mediante interpretación, realizando todas las comprobaciones de tipo de forma dinámica, donde podremos manipularlas si es necesario. El modelo de ejecución del estándar *CLI* no presenta ningún problema en este sentido.
- La posibilidad de acceder a todo el entorno computacional del motor de ejecución desde el código que controle el proceso de traducción de instrucciones del lenguaje intermedio a nativo y también desde el propio código nativo. Si todo o parte de la metainformación del programa desapareciese del motor de ejecución al ejecutarse un programa, no seria posible implementar todos los procesos necesarios para integrar primitivas reflectivas en el sistema. La existencia de capacidades de introspección en el sistema base nos asegura que esto no ocurrirá (ya que en otro caso no podrían ser implementadas).

Por tanto, debemos modificar la forma en la que el compilador *JIT* traduce los accesos a miembros actualmente, representada en la figura 9.2, a una nueva forma que sea válida para usar las nuevas capacidades de reflexión (accediendo a la información dinámica añadida) desde el código nativo que se genere, representado en la figura 9.3.

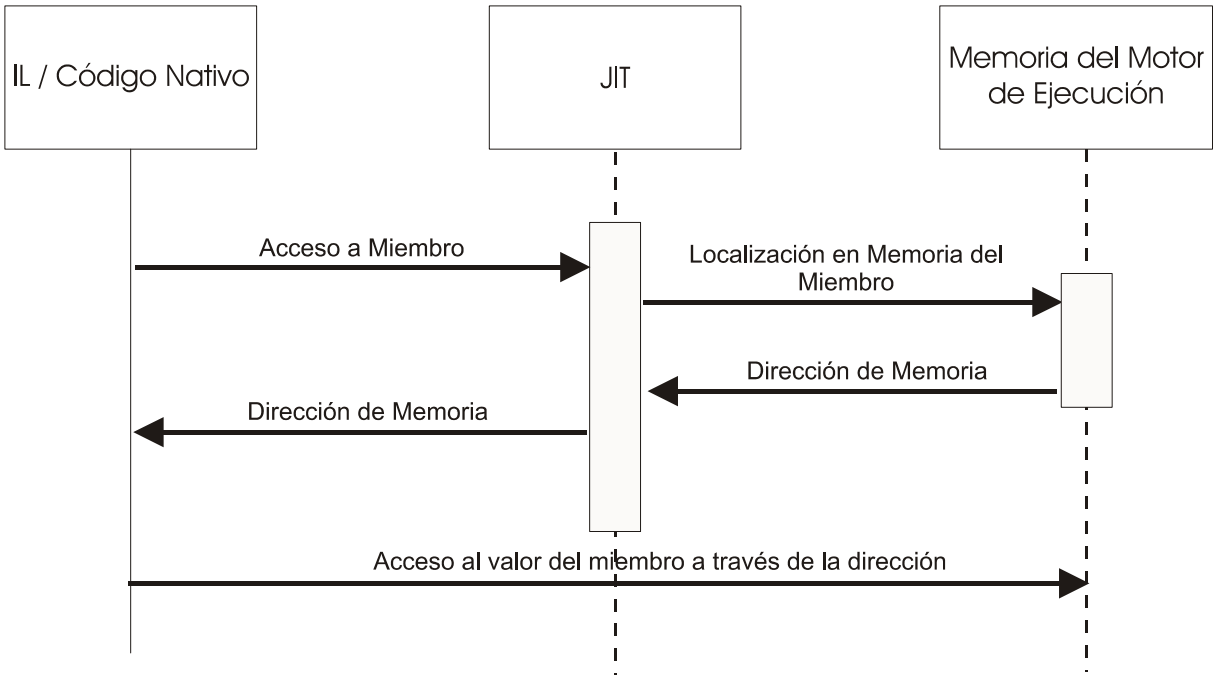

Figura 9.2: Proceso de conversión de un acceso a un miembro en *CIL* en código nativo en el *JIT* original

En este diagrama podemos ver cómo un compilador *JIT* "estándar", como el que

tendrá el sistema base sin modificar, convertiría un acceso a un miembro cualquiera en código intermedio en una dirección de memoria física a la que acceder o llamar desde código nativo, desde la cual se pueda hacer uso del valor que contiene, maximizando de esta forma el rendimiento en ejecución (al no tener que efectuar indirecciones) del código nativo generado. Este procedimiento es adecuado para lenguajes estáticos, donde se puede tener la seguridad de que el miembro no será alterado y que, una vez cargada la información del programa en memoria, ésta permanecerá allí con el formato y estructura conocidos en tiempo de compilación. Evidentemente, esta suposición no podrá ser empleada con lenguajes dinámicos, y el mecanismo deberá ser transformado de la forma que se muestra en la figura 9.3 para poder implementar operaciones reflectivas.

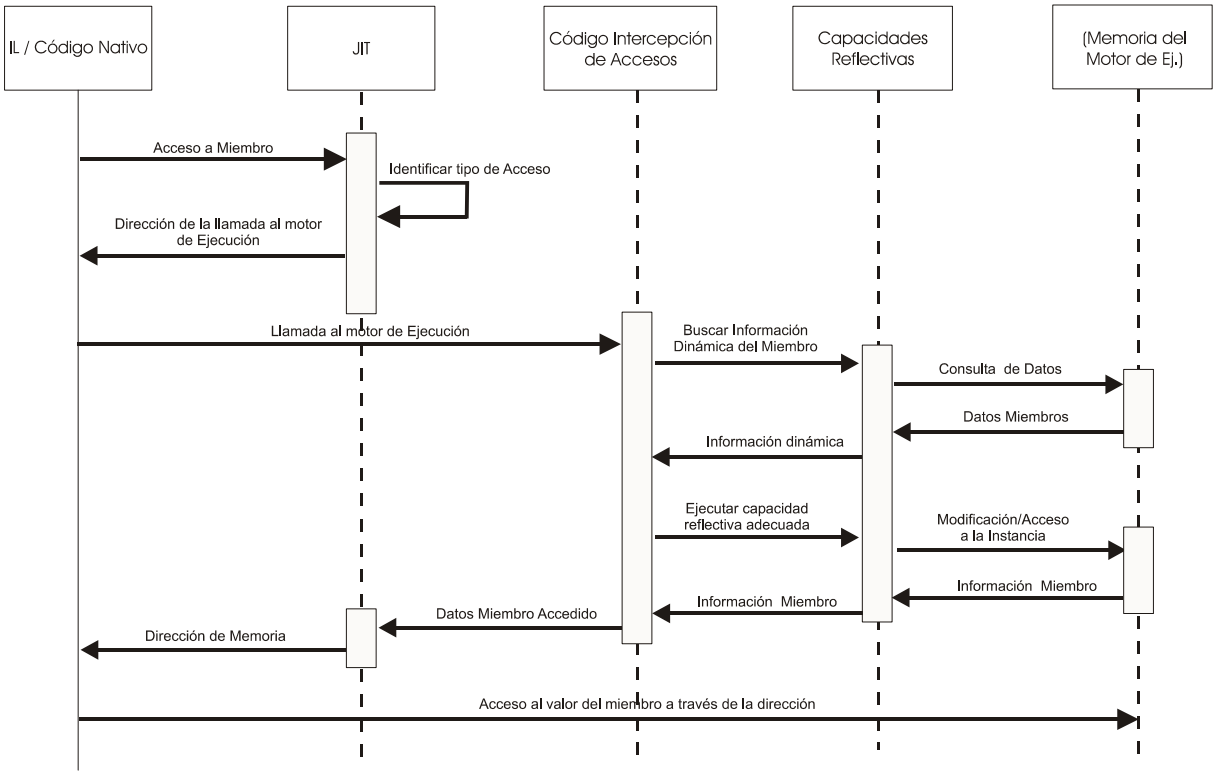

Figura 9.3: Proceso de conversión de un acceso a un miembro en *CIL* en código nativo en un *JIT* modificado

En este segundo diagrama podemos apreciar cómo para acceder al miembro deseado ya no se podrá acceder directamente a unas zonas de memoria determinadas del motor de ejecución, sino que el código nativo generado por el *JIT* incluirá una llamada al motor de ejecución. Esta llamada buscará entre toda la información (añadida o existente) del propio motor de ejecución para ver si ésta ha sufrido algún tipo de modificación que afecte a la operación que se está llevando a cabo en ese momento. En caso de que sí la haya, posteriormente se deberá ejecutar la capacidad reflectiva adecuada de la manera que sea necesaria según el estado actual del sistema, de forma que sólo después de que se realicen estas operaciones se podrá devolver una dirección de memoria que se pueda usar para acceder al valor que se necesita en el código nativo. En caso de que no exista información que afecte al proceso, se puede devolver la misma información que con el *JIT* sin modificar y continuar el proceso normalmente. Como puede verse, la responsabilidad de hacer las búsquedas pertinentes será ahora de un código especial de intercepción de accesos, una entidad abstracta que se ha pensado que podría encapsular este tipo de operaciones nuevas en la medida de lo posible.

En definitiva, el mecanismo original de acceso a miembros permitía al *JIT* acceder a la información necesaria para la transformación de código, capacitándole para localizar

aquella que necesitaba de manera muy eficiente. Este mecanismo se apoyaba en que el tamaño de todos los tipos e instancias y la localización de cualquiera de sus miembros ya era conocida e inamovible después del proceso de compilación inicial. Dado que estas suposiciones no son válidas cuando se utiliza reflexión estructural, el proceso del *JIT* se tendrá que transformar ahora en una operación de búsqueda más compleja, que permita al código nativo generado localizar información que pueda haber sido cambiada por el propio programa en su ejecución, y que por tanto ya no pueda suponerse su localización en una zona de memoria predeterminada.

Vemos pues cómo el motor de ejecución tiene que verse alterado no sólo añadiéndole más elementos, sino que el proceso normal de tratamiento de los programas que se ejecutan con el mismo también requerirá modificaciones para tener en cuenta la nueva situación producida por los añadidos que se le van a realizar. Sólo se han presentado de forma abstracta estos cambios a nivel de diseño dentro de la estructura general del propio motor, y tendrán que ser matizados y detallados a la hora de crearlos sobre la implementación del estándar *CLI* que finalmente sea escogida, adaptándolos a las peculiaridades del motor de esa implementación. La figura 9.4 ilustraría la disposición deseada de los elementos involucrados en la secuencia mostrada en la figura anterior sobre la implementación del *CLI* que se escoja. Aquí se ve cómo el *JIT* genera el código nativo de forma que éste tenga que acceder al motor de ejecución ante un acceso a un miembro, empleando las capacidades reflectivas implementadas para acceder a cada instancia y consultar la información añadida que pudiese poseer. Este proceso pues completa las modificaciones que debemos hacer al *JIT* del motor de ejecución para los fines propuestos.

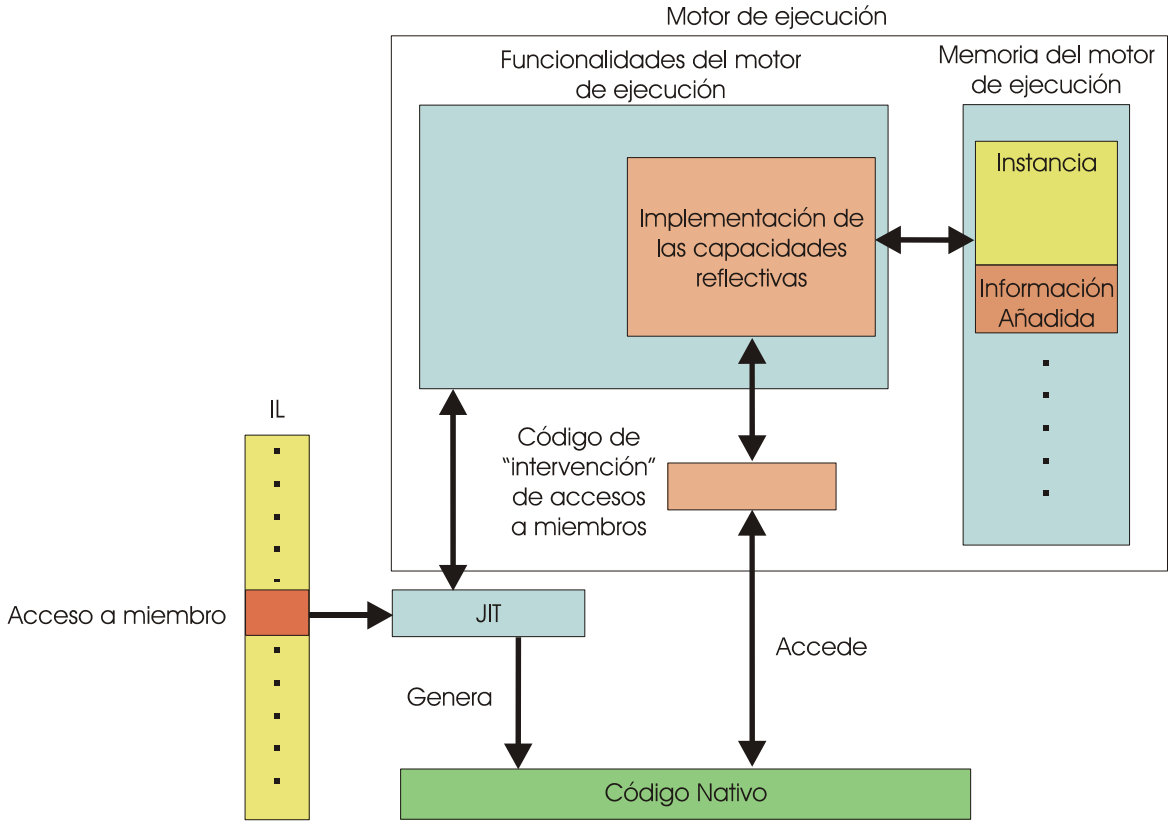

Figura 9.4: Diseño del acceso desde *CIL* a capacidades reflectivas

9.3 EL SISTEMA CLI

Aunque la máquina virtual *CLI* ha sido escogida por cumplir gran parte de los requisitos planteados en esta tesis, este sistema necesita modificaciones para llevar a cabo todos los objetivos pretendidos. Sin duda uno de los mayores problemas con los que nos encontraremos será dotarle de un soporte eficiente para los dos tipos de lenguajes que se pretenden implementar sobre el mismo, ya que está únicamente preparado y optimizado para la ejecución de lenguajes estáticos basados en clases. En este apartado haremos una enumeración somera de los principales problemas que encontraremos en el *CLI* al respecto y cómo deberán diseñarse las modificaciones del motor de ejecución para integrar estos elementos en el sistema final. La integración concreta de las funcionalidades descritas dentro de la máquina se detallará en un capítulo posterior.

9.3.1 Adaptación del CLI

Existen varios aspectos del diseño del motor de ejecución de *CLI* original que deben modificarse para dar un soporte adecuado a primitivas reflectivas de lenguajes dinámicos, ya que en su estado actual no son adecuados para este fin. A continuación enumeraremos aquéllos más relevantes:

- **La comprobación de tipos**: Gran parte de las operaciones de comprobación de tipo de los datos en general dentro de la máquina deberá realizarse en tiempo de ejecución. Los lenguajes de alto nivel serán los encargados de decidir si realizan o no comprobaciones de tipo cuando su código se transforma en *CIL* en función del tipo de lenguaje pero, para permitir la existencia tanto de lenguajes estáticos como de lenguajes dinámicos, es necesario que cuando este *CIL* vaya a transformarse a código nativo mediante la compilación *JIT* no se hagan comprobaciones estáticas, ya que sino no será posible proporcionar soporte para el segundo tipo de lenguajes. Por tanto, se deberán modificar o alterar todas las comprobaciones que se realicen cuando se transforma el código intermedio.
- **Primitivas reflectivas**: El motor de ejecución debe ser ampliado para integrar las nuevas primitivas reflectivas como una funcionalidad estándar más, pudiendo acceder a ellas de la misma forma que el resto de las funcionalidades del motor.
- **La estructura de clases e instancias**: Una máquina, diseñada y optimizada para ofrecer un soporte lo más eficiente posible a lenguajes estáticos, no contempla en su diseño que sus clases o instancias puedan alterar dinámicamente su estructura y por tanto puedan tener un tamaño variable. Normalmente supondrá que estas entidades son de tamaño fijo y optimizará el código basándose en ello. Se debe modificar el motor de ejecución para que clases e instancias puedan albergar nuevos miembros y tengan un tamaño variable en ejecución.
- **El acceso a miembros**: Como hemos visto, una máquina diseñada para ofrecer la máxima eficiencia con lenguajes estáticos generará un código nativo que accederá directamente a los miembros de clases o instancias que necesite, evitando indirecciones innecesarias y maximizando así el rendimiento de estas operaciones, que normalmente consistirán en un simple acceso a una zona de memoria concreta. Un lenguaje dinámico no puede tener la misma aproximación que uno estático en este aspecto, ya que es posible que un miembro deje de existir en

cualquier momento por haber sido eliminado o modificado, y transformar un acceso al mismo en una dirección de memoria acarrearía problemas. Por ello, el acceso a miembros deberá modificarse para contemplar accesos simbólicos (por nombre) cuando sea necesario, de manera que el código nativo sea capaz de calcular dinámicamente donde se encontrará un miembro en un instante dado.

- **Semántica del juego de instrucciones**: El acceso a las capacidades reflectivas que se van a implementar debe realizarse de forma que, sin alterar los lenguajes ya existentes de la máquina abstracta, estén disponibles para cualquiera de ellos además de para cualquier lenguaje nuevo que se añada (dinámico o no). La mejor forma de permitir el uso de capacidades reflectivas, sin añadir nueva sintaxis a ningún lenguaje, es dotar a la librería base de clases del sistema (*BCL*) de nuevas clases y objetos que modelen estas funcionalidades y que puedan emplearse de la misma forma que otras clases de la misma. No obstante, mantener este tipo de acceso para todos los lenguajes no será viable dado que tendrá un impacto evidente en el rendimiento ofrecido por estas primitivas, por lo que para permitir una implementación más eficiente de nuevos lenguajes que las usen (dinámicos, por ejemplo) se deberá alterar la semántica de algunas instrucciones del lenguaje intermedio de la máquina para que contemplen las nuevas capacidades reflectivas en su procesamiento, y que de esta forma se permita el uso de estas primitivas de una forma más optimizada, ya que así los lenguajes nuevos que se añadan a la máquina, que no tendrán ninguna restricción sintáctica al diseñarse ya en torno a estas primitivas, podrán implementarse generando código *CIL* directamente.

La figura 9.5 ilustra lo comentado en el párrafo anterior. Los lenguajes ya existentes (líneas continuas) podrán acceder a la implementación de las primitivas a través de clases que las representen en la *BCL*, compilándose posteriormente a código *CIL* estándar de la máquina cuyo comportamiento será análogo al del sistema base. Los lenguajes dinámicos nuevos (líneas de puntos) podrán compilarse directamente a código *CIL* cuya semántica modificada le permitirá acceder a las nuevas capacidades reflectivas implementadas (figura 9.3). En ambos casos, a pesar de acceder a las primitivas desde diferentes puntos, su implementación dentro del motor de ejecución es compartida.

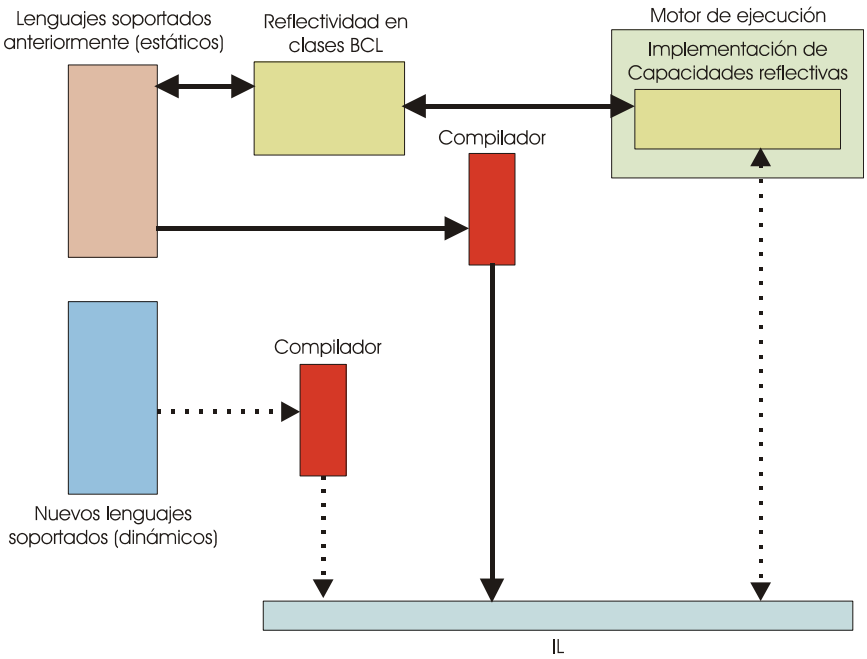

Figura 9.5: Integración de nuevas funcionalidades en el sistema

No obstante, no todas las operaciones encajarán dentro de este modelo. Como no deseamos modificar ningún lenguaje existente en la máquina (incluido el *CIL*) de cara a mantener una total compatibilidad hacia atrás, para hacer determinadas operaciones (añadir miembros, por ejemplo) se tendrá que usar una función de la librería, al no poder crearse una instrucción *CIL* específica para ello. La figura 9.6 muestra más en detalle la disposición propuesta de los elementos involucrados en el acceso desde un programa cualquiera a una capacidad reflectiva implementada a través de una función *BCL*. Podemos comprobar cómo cualquier programa de alto nivel accederá a una capacidad reflectiva a través de la llamada a una funcionalidad de la *BCL* que realmente se ejecutará dentro del motor de ejecución. Ésta accederá a las instancias y la posible información añadida a las mismas. El programa será compilado normalmente y el acceso a capacidades reflectivas se traducirá a llamadas a métodos estándar.

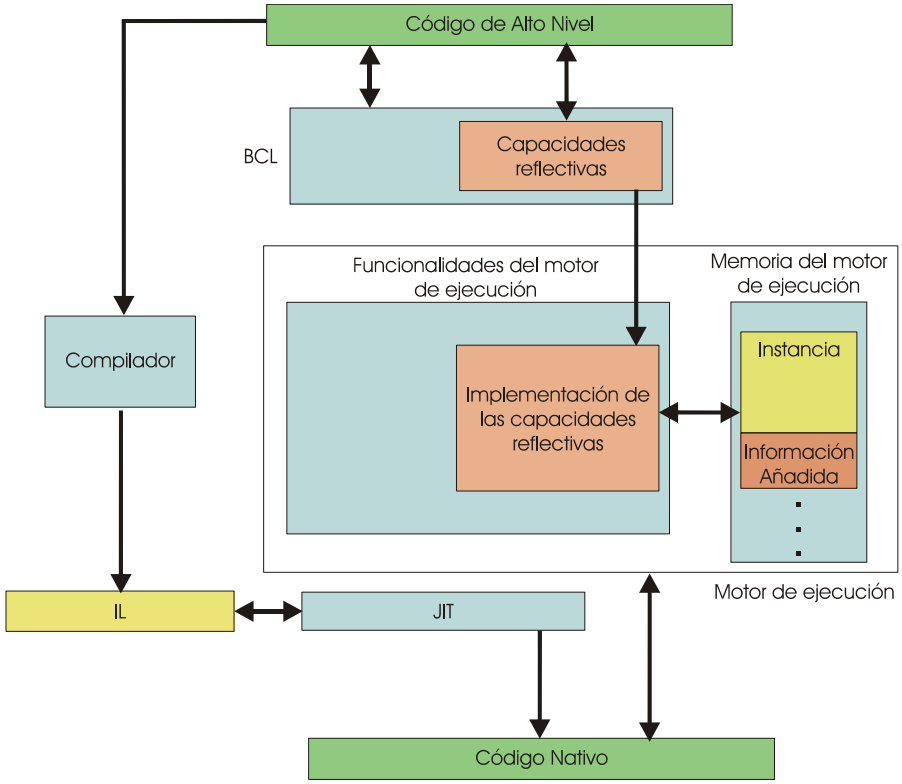

Figura 9.6: Acceso de un programa a capacidades reflectivas

- **Creación y manipulación de nuevos miembros**: A la hora de implementar las capacidades reflectivas, cuando se necesiten añadir nuevos miembros a los objetos el programador debe ser capaz de crearlos desde cero y configurar todas sus características. Gracias a que la máquina abstracta original tiene soporte para introspección, ya existen clases que son capaces de representar todos los metadatos de atributos y métodos, por lo que sólo tendremos que usarlas (o bien hacer derivados de las mismas, por si es necesario guardar alguna información adicional) para ello. Por tanto, las estructuras de datos que permitirán crear y manipular nuevos miembros deberán ser compatibles con las ya existentes.
- **Modelo computacional**: Como ya se ha mencionado anteriormente, el modelo computacional original de *CLI* no está preparado para soportar las características de los lenguajes dinámicos y deberá ser modificado para ello. En el siguiente capítulo se detallará esta transformación del modelo.

10 MODELO COMPUTACIONAL

10.1 REFLEXIÓN ESTRUCTURAL VS. REFLEXIÓN COMPUTACIONAL

Como ya se ha mencionado en el capítulo dedicado a la arquitectura del sistema, se ha tomado la decisión de incorporar reflexión estructural sobre la máquina abstracta en detrimento de la reflexión computacional. Esta decisión ha estado motivada por varias razones, que detallaremos en profundidad a continuación:

- **Rendimiento**: Dada la complejidad inherente de los mecanismos que deberán ser implementados en el caso de querer incorporar reflexión computacional (ya que se debe permitir modificar la semántica de un determinado conjunto de primitivas del sistema), se ha considerado que tanto la penalización de rendimiento como el coste de implementación que ello supondría sería demasiado elevado para los beneficios de flexibilidad que se obtendrían.
- **Nivel de flexibilidad**: El nivel de flexibilidad proporcionado por la reflexión computacional es sin lugar a dudas superior al proporcionado por la estructural. No obstante, este último es lo suficientemente potente como para permitir alcanzar un alto nivel de flexibilidad, adecuado para la mayoría de aplicaciones que necesiten responder ante cambios en su modelo de negocio y/o requisitos, hacia las cuales está orientado este trabajo.
- **Flexibilidad de Lenguajes Dinámicos**: La mayoría de los lenguajes dinámicos vistos (*Python*, *Smalltalk*, *Self*, etc.) incorporan en su modelo computacional mecanismos de flexibilidad mediante reflexión estructural, y sólo se permiten en algunos casos operaciones relativas a reflexión computacional de forma limitada con ciertas primitivas. Esto refuerza la idea de que este nivel de reflexión es adecuado para las aplicaciones que necesiten un determinado nivel de flexibilidad para sus fines.
- **Limitaciones de funcionalidad**: La pérdida de flexibilidad provocada por la implementación de un nivel menor de reflexión es menor de la que en principio puede pensarse, dado que es posible emular ciertas operaciones propias de reflexión computacional empleando de forma combinada construcciones de un nivel de reflexión inferior [Sun99], tal y como se vio anteriormente.
- **Limitaciones de la arquitectura base**: Dado que no se va a crear una máquina desde cero que soporte las primitivas reflectivas, y se parte de una no preparada inicialmente para el soporte directo de las mismas, los cambios que esta arquitectura requerirá para ello serán de una complejidad muy elevada, considerándose pues más sencillo y viable ofrecer características de reflexión estructural, que por otro lado parecen ser más viables teniendo en cuenta las consideraciones acerca del diseño del motor de ejecución vistas en el capítulo anterior.
- **Estudio de los sistemas existentes**: Como se ha visto en el estado del arte, la

mayoría de sistemas dotados de reflexión computacional la ofrecen de una forma limitada, estableciendo de antemano qué partes se podrán modificar de esta manera. El soporte de este tipo de reflexión en muchas ocasiones requerirá la implementación de una gran cantidad de código de soporte adicional, como protocolos de metaobjetos (*MOPs*) o similares, con un gran impacto en el rendimiento de las aplicaciones. Por ello, dada la práctica inexistencia de sistemas que ofrezcan un soporte de reflexión computacional completo que ofrezca un rendimiento adecuado, se ha decidido realizar el estudio propuesto sobre un nivel de reflexión que sí tiene una serie de sistemas asociados de rendimiento aceptable.

Todas las razones anteriores justifican la decisión tomada de implementar un nivel de reflexión estructural frente a uno computacional. Según el análisis realizado, la reflexión estructural logra el mejor balance posible entre el rendimiento ofrecido y la flexibilidad obtenida, permitiendo gozar de un grado de flexibilidad elevado y suficiente para la mayoría de necesidades sin que potencialmente se deba sacrificar excesivamente el rendimiento del sistema.

A pesar de lo dicho, hemos visto sin embargo cómo la implementación de reflexión estructural sobre una máquina abstracta estática existente no está exenta de sus propios problemas, tanto arquitectónicos como de modelo computacional. Mientras en el capítulo anterior hemos visto cómo los problemas arquitectónicos pueden superarse mediante un estudio de la arquitectura de la que disponemos, es necesario hacer una reflexión más en detalle acerca del modelo computacional que soportará todas estas operaciones, algo que haremos en el capítulo siguiente.

10.2 APLICACIÓN DE REFLEXIÓN ESTRUCTURAL A MODELOS BASADOS EN CLASES

10.2.1 Inconsistencias con el Modelo Basado en Clases

A la hora de incorporar capacidades de reflexión estructural a un modelo computacional de orientación a objetos basado en clases existen ciertos problemas conceptuales que se deben tratar adecuadamente para que el modelo resultante sea coherente. Las inconsistencias que vamos a describir a continuación fueron ya detectadas en el campo de las bases de datos orientadas a objetos, existiendo soluciones de compromiso que no resuelven en su totalidad los problemas planteados.

10.2.1.1 EVOLUCIÓN DEL ESQUEMA

En una base de datos, cuando se guardan objetos para un uso posterior, se espera que tanto su estructura como su tipo (clase) puedan ser modificadas posteriormente según las necesidades del *software* que trabaja con ellos [Skarra87]. Si tenemos una serie de objetos guardados, y necesitamos modificar la estructura de su tipo (añadiéndole atributos, por ejemplo), entonces será necesario alterar de forma consecuente la composición de todo objeto que sea instancia de dicha clase o tipo. Esto es debido a que en un modelo basado en clases se supone que es la clase la que debe reflejar tanto la estructura como el comportamiento de todos sus objetos, por lo que alterar una clase, sin modificar convenientemente todos los objetos vinculados con ella, violaría un principio fundamental de este modelo. En el mencionado campo de las bases de datos, al mecanismo que permite modificar los objetos de manera acorde a los cambios que se efectúan sobre sus clases se le conoce como evolución de esquemas (*schema evolution*) [Roddick95].

La modificación de las instancias de una clase concreta en este escenario se puede realizar en el momento en que la clase es modificada (de forma ansiosa) o bien cuando el objeto va a ser usado (de forma perezosa) [Tan89]. Para ello sólo es necesario saber la clase de la que un objeto es instancia en un momento dado en tiempo de ejecución. De la primera forma, en el momento en que se haga una modificación cualquiera a una clase, se deberán buscar todas y cada una de las instancias activas de la misma y efectuar una modificación en ellas que refleje la que se ha hecho en su clase. El segundo método implicaría, sin embargo, que si se trata de acceder a un miembro concreto en una instancia que no está registrado en la estructura de la misma, antes de devolver una excepción notificando el error, el sistema accederá a su clase y buscará dicho miembro, de manera que si lo encuentra la operación no devuelva un error, sino que incorporará los cambios a la instancia y la operación terminará normalmente. Esto implica que los cambios hechos a las clases sólo se reflejarán en sus instancias si éstas hacen uso de estos cambios. El mismo mecanismo se seguirá en cualquier operación de acceso a miembros para mantener en todo momento la coherencia instancia - clase.

Se deben no obstante hacer ciertas consideraciones acerca de este tipo de operaciones. Aunque ambos métodos de actualización descritos son igualmente válidos para conseguir los fines propuestos, está claro que su forma de operar tiene grandes diferencias que podrán tener un impacto diferente sobre el sistema. La búsqueda activa de instancias pertenecientes a una clase es problemática por varios motivos:

- Es posible que no tengamos esta información o ésta sea muy costosa de obtener. En un sistema orientado a objetos basado en clases es necesario que a partir de una instancia se pueda obtener su clase, pero no es frecuente que una clase pueda acceder a una lista completa de sus instancias en un instante dado fácilmente. Tampoco es viable recorrer cada instancia existente en un momento dado para obtener su clase, y recopilar así el conjunto de instancias modificables en un momento dado, por el coste y la complejidad que esta operación supondría.
- Aun poseyendo una forma eficiente de obtener todas las instancias de una clase, si el estado de la ejecución actual implica la existencia de una gran número de instancias en memoria en un momento dado, el coste de hacer las modificaciones necesarias se podría disparar.
- Tratar de modificar todas las instancias ante cualquier tipo de modificación de una clase también incrementaría enormemente el coste de esta operación. Una simple modificación de carácter temporal introduciría la misma complejidad que otra que tuviese un carácter más definitivo, el sistema no podría distinguir entre ambas.
- Por último, es razonablemente posible asumir que a pesar de que una clase se modifique con la idea de incorporarle soporte para una determinada funcionalidad,

esta nueva funcionalidad no sea usada por todas sus instancias. Esto implica que se estaría malgastando la memoria principal del sistema, puesto que los cambios a cada instancia se harán independientemente de si son o no necesarios.

Estos motivos desaconsejan pues la primera política de actualización de instancias, dejando claro que la elección de la segunda es más adecuada, sobre todo de cara a obtener un mejor rendimiento, lo que es un objetivo primordial en esta tesis. Si el sistema sólo actualiza una instancia cuando ésta muestra interés en usar las modificaciones realizadas a su clase, entonces en ningún momento se estarán malgastando recursos (ni de tiempo ni de memoria), ya que sólo se consumirán con elementos que vayan a ser usados en un momento dado. Por otra parte, si se hace una modificación a una clase de carácter temporal que posteriormente es eliminada, el impacto en rendimiento de estas modificaciones será prácticamente inexistente. No obstante, la evolución dinámica de los métodos y atributos de una clase puede ocasionar que se acceda en tiempo de ejecución a atributos o métodos que no existan en un momento dado, pero este tipo de situaciones pueden ser controladas implementando un mecanismo de manejo de excepciones adecuado y empleando las comprobaciones de tipo dinámicas necesarias para asegurar que no se producen resultados inesperados o incorrectos.

La solución al problema descrita puede ser aplicada directamente tanto en bases de datos como en un programa que siga el modelo de orientación a objetos basado en clases, suponiendo que su lenguaje permite modificar la estructura y/o comportamiento de sus clases (reflexión estructural). No obstante, aunque el problema anteriormente expuesto ofrece una solución viable, existe un problema adicional cuya solución es mucho más compleja. Este segundo problema aparece cuando se intenta modificar la estructura de una instancia concreta en lugar de su clase.

10.2.1.2 VERSIONADO DE ESQUEMAS

Si se trata de modificar únicamente una sola instancia, sin alterar ninguna de sus instancias "hermanas" ni la clase a la que pertenecen, estaríamos rompiendo con las restricciones del modelo de objetos basado en clases, al crear una instancia cuya estructura ya no estaría reflejada por su clase. Como ya se ha visto en el estado del arte de esta tesis, este problema fue detectado en el desarrollo del sistema *Metaxa* [Golm97c], y la solución adoptada en el mismo también refleja la empleada por algunos sistemas de bases de datos orientados a objetos, denominada versionado de esquemas. Como se vio anteriormente, *Metaxa* crea una clase adicional para estas instancias modificadas denominada "clase sombra" (*shadow class*) que refleje la estructura de esa instancia particular. No obstante, hemos visto anteriormente cómo esta técnica plantea una serie de inconvenientes [Golm97c] relativos a la consistencia de atributos entre clase original y clase sombra, la identidad de las clases (clase original y sombra deben representar el mismo tipo), con los objetos del sistema que representan a las clases (ahora habrá dos objetos involucrados en las operaciones sobre las clases), la recolección de basura (cuando borrar una clase sombra), la consistencia del código, el mayor consumo de memoria y la herencia (una clase sombra a una clase C dada necesita también otras clases sombra de las clases padres de C). Los numerosos problemas que causa esta solución hacen que no la consideremos adecuada para los fines que se persiguen en esta tesis, dada la complejidad que introduce [Golm97c] y su escasa viabilidad.

Además de los dos problemas anteriores, debemos tener en cuenta que la fuerte relación y dependencia existente entre una clase y sus instancias, en el modelo de objetos basado en clases, hace que implementar soporte para mecanismos de herencia basada en una estrategia de delegación y el cambio dinámico de tipo de una instancia

concreta, ambas necesarias para modelar correctamente una base computacional que soporte lenguajes dinámicos, resulte más complejo.

Por tanto, dado que el modelo basado en clases no encaja adecuadamente con las capacidades de reflexión estructural que se pretenden implementar, se debe buscar una solución alternativa para permitir su implementación. Para ello, es necesario usar los principios del modelo de orientación a objetos basada en prototipos visto en la primera parte de esta tesis, para crear un modelo mixto que permita al mismo tiempo:

- La ejecución de lenguajes estáticos y dinámicos con un modelo computacional coherente con las capacidades de ambos, que permita hacer todas las operaciones necesarias, incluyendo un soporte completo de reflexión estructural.
- La cooperación de ambos modelos para permitir la interoperabilidad entre ambos tipos de lenguajes.
- Conservar la total compatibilidad con el modelo anteriormente existente, de forma que todo el código heredado se pueda ejecutar sobre la máquina extendida.

Con estos propósitos en mente, se ha creado un modelo computacional alternativo que se describirá en la siguiente sección.

10.2.2 Modelo Computacional Propuesto

Para crear un modelo computacional que cumpla con las características deseadas se han usado conceptos del modelo de prototipos ya descrito, de manera que se puedan solventar los problemas ocasionados por el versionado y la evolución de esquemas. Como ya se ha dicho, en el modelo de prototipos la principal abstracción es el objeto, y no existe el concepto de clase como tal. El nuevo modelo computacional girará en torno a los principios definidos en el capítulo dedicado al modelo de prototipos:

- El comportamiento compartido por todos los objetos (por ejemplo, los métodos de una clase en el modelo basado en clases) se representa mediante la creación de un objeto *trait*.
- La estructura compartida por los objetos (los atributos de una clase en el modelo basado en clases) puede representarse mediante la creación de objetos prototipo. Estos prototipos contendrán todo el conjunto de atributos de cada instancia.
- La creación de nuevas instancias se hace mediante la clonación de los objetos prototipo, proceso que sería equivalente a la creación de un objeto por medio de su constructor en el modelo de clases.
- El mecanismo de herencia usado por los lenguajes dinámicos, que como hemos visto en un capítulo anterior está basado en delegación, a diferencia del mecanismo de concatenación usado por los lenguajes estáticos, determinará la forma de localizar e invocar métodos en dicho modelo.

Ha de tenerse en cuenta que el uso del modelo de prototipos no supone una pérdida de expresividad respecto al de clases en ningún caso [Wolczko96]. No obstante, la principal ventaja que tiene este modelo es que permite implementar fácilmente las operaciones reflectivas que necesitamos:

- El modelo elimina el problema del versionado de esquemas, ya que ahora se puede modificar directamente cualquier miembro de un objeto aislado, al ser éste el encargado de mantener su propia estructura y comportamiento particular. La no existencia de clases elimina vinculaciones estrictas de las instancias con elementos ajenos a ella, siendo pues ellas mismas responsables de su propio comportamiento y estructura. Por tanto, toda instancia podrá adquirir nuevos miembros locales, de las que ella misma es propietaria, sin restricciones de ningún tipo.
- Todo aquello que sea compartido por una serie de objetos podrá agruparse en un objeto *trait*, elemento que, como hemos visto, contendría métodos que pueden ser usados por un conjunto de instancias asociado. Modificar un objeto *trait* hace que todos los objetos asociados al mismo vean automáticamente esta modificación en el momento que se produzca, al delegarle estos objetos, gracias al mecanismo de herencia empleado, las llamadas a métodos que no sean capaces de resolver ellos mismos.
- El mecanismo de herencia por delegación asegura que toda relación entre instancias y/o *trait objects* pueda ser dinámica, de manera que si se invoca un método sobre un objeto y éste no posee dicho método, se hará una búsqueda recorriendo todos los objetos que estén relacionados con dicha instancia en ese momento, actuando en consecuencia. Si una instancia está asociada jerárquicamente con N *trait objects*, se hará una búsqueda por todos ellos para encontrar el método pedido. Como las relaciones instancia - *trait object* son dinámicas, ninguna instancia estará relacionada con un conjunto de métodos fijo y se permitirá una mayor flexibilidad.

Basándonos en todos los conceptos vistos, el nuevo modelo computacional quedará configurado como se explicará a continuación. Ha de tenerse en cuenta que, puesto que debemos mantener la compatibilidad en todo momento, el modelo de clases existente no desaparece y de hecho estará activo en su forma actual cuando los lenguajes no requieran del uso de capacidades de reflexión. Es pues cuando se haga uso de alguna de las nuevas primitivas reflectivas cuando el nuevo modelo computacional se comportará de la manera que se va a describir, proporcionando de forma efectiva soporte para lenguajes dinámicos creando un modelo computacional nuevo que funcione sobre el modelo basado en clases existente. Las clases se modificarán usando para ello el objeto que el sistema nos proporciona para acceder a sus metadatos (`System.Type`). Por todo ello, en el nuevo modelo computacional:

- Modificar los métodos de un `Type` sería equivalente a modificar la adaptación de un comportamiento compartido y por tanto sería idéntico a modificar un objeto *trait object,* tal y como hemos descrito anteriormente. Dado que en el modelo de clases el comportamiento es propiedad de las clases, deberemos simplemente asegurarnos de que a la hora de invocar algún método añadido sobre alguna instancia, el mecanismo de búsqueda tiene en cuenta este hecho (herencia por delegación), modificando el motor de ejecución adecuadamente para ello. Esta operación ocurre tanto en el modelo de clases como en el modelo de prototipos soportado.
- Si lo que pretendemos es modificar los atributos de una clase, entonces se provocará una modificación acorde de todas las instancias existentes de esa clase. Esto causa nuevamente una evolución de esquemas como la mencionada anteriormente, cuya solución es implementable mediante el mismo mecanismo anterior siguiendo los conceptos ya vistos, usando un mecanismo perezoso para la evolución de esquemas [Tan89]. Esta operación sólo se utilizaría en el modelo de clases.

- Ante una operación reflectiva sobre una instancia, los objetos son tratados como objetos en el modelo basado en prototipos. De esta forma, a cualquier objeto podrán añadírsele o eliminársele tanto métodos como atributos directamente. Dado que un lenguaje estático no usará las capacidades de reflexión, permitir añadir información adicional a las instancias no les afectará (ya que para ellos simplemente no existe). Es por tanto el lenguaje de programación de alto nivel quien decide qué modelo computacional desea usar, ya que el sistema permite soportar ambos de forma consistente. Esto permite proporcionar soporte para lenguajes dinámicos orientados a prototipos sin alterar el soporte para lenguajes estáticos, puesto que un compilador dedicado a uno de estos lenguajes (como *C#*) no usará las capacidades reflectivas debido a que espera encontrarse con el modelo de clases estático "tradicional", que es alterado para permitir modificar la estructura de sus elementos.

Por tanto, vemos cómo es posible crear un modelo computacional que permita convivir a lenguajes dinámicos y lenguajes estáticos bajo una misma máquina, de manera que los lenguajes estáticos ya existentes vean sólo la parte del modelo que requieren (la existente originalmente) mientras que se proporciona un modelo alternativo para aquellos lenguajes que necesiten usar capacidades reflectivas, modelo que es similar al implementado por lenguajes dinámicos ya existentes y que está basado en prototipos, permitiendo de esta forma emplear todas las primitivas reflectivas implementadas sin caer en incongruencias. También ha de tenerse en cuenta la posibilidad que este nuevo modelo computacional proporciona de soportar lenguajes que, estando basados en clases, permitan reflexión estructural a nivel de clases solamente, al estar soportado un mecanismo de evolución de esquemas sobre las instancias que permite actualizar las mismas adecuadamente de acuerdo a las modificaciones realizadas.

Todo lo presentado además ha sido creado partiendo de un modelo de clases, por lo que alterar el soporte del mismo de la máquina abstracta para dotarle de las nuevas funcionalidades parece viable.

10.3 DISEÑO DE LAS PRIMITIVAS REFLECTIVAS OFRECIDAS

Una vez descritos los principios del modelo computacional que va a ser usado por la máquina extendida, se podrá detallar el diseño de las operaciones reflectivas a implementar [Redondo06] [Redondo07]. No entraremos en ningún tema relativo a la implementación, ya que aún no se ha elegido la distribución concreta del *CLI* que se va a emplear para ello, sino que describiremos su funcionamiento e interacción prevista con otras partes del sistema.

Ante todo debemos tener en cuenta que para implementar las primitivas reflectivas no se va a modificar el lenguaje intermedio de la máquina, algo sobre lo que ya se ha insistido anteriormente. El principal motivo es evitar incompatibilidades con el *software* ya desarrollado que no sean evidentes a primera vista (los cambios introducidos podrían ocasionar efectos laterales no deseables), asegurándose una máxima compatibilidad. Por ello, como ya se mencionó en el apartado de diseño del motor de ejecución, las primitivas reflectivas implementadas residirán en el motor de ejecución y se podrá acceder a ellas por dos vías diferentes. En este apartado no analizaremos las vías de acceso a las mismas (ya lo fueron en la parte de diseño), sino la acción a llevar a

cabo por cada servicio concreto al que se accede.

Por tanto, se enumerarán a continuación todas las primitivas a implementar, correspondientes a la reflexión estructural seleccionada, y se hará una descripción de las operaciones a realizar por cada uno de ellos. También se mostrarán diagramas de algunas operaciones para completar la explicación de lo que cada una de ellas debe hacer dentro del sistema donde sea necesario. Al encontrarnos en una fase de diseño y no tener hasta el momento información acerca de las características concretas de la máquina que se va a usar como punto de partida (puesto que las implementaciones del estándar *CLI* pueden variar en lo referente a su arquitectura interna, aun representando al mismo estándar), los diagramas que se presentarán a continuación (y en general todos los existentes en este capítulo) han sido creados con un nivel de abstracción más elevado, teniendo sólo en cuenta los elementos que podemos encontrar a partir del documento que especifica el estándar o bien aquéllos que, tras hacer un estudio de las posibles implementaciones existentes, sabemos que podremos encontrar en todas ellas.

10.3.1 Operaciones sobre Atributos

Se describirán a continuación todas las operaciones cuya implementación está prevista para trabajar con los atributos de una clase o instancia:

- **Obtener un atributo**: Este servicio será el encargado de obtener un atributo y devolverlo al usuario, de forma que no se pueda distinguir entre atributos que han sido añadidos en tiempo de ejecución y aquéllos que se incorporen por el código del programa en sí. Esto debe hacerse usando una representación idéntica o muy similar para ambos tipos de atributos.

  Además, debe tenerse en cuenta la posibilidad de que ese atributo al que se accede en un momento dado haya podido ser eliminado dinámicamente, devolviendo la excepción correspondiente en ese caso para que el usuario pueda controlar la incidencia y tratarla en su programa. Para dotar a ambas clases de atributos de una representación homogénea se usará la clase `FieldInfo` (o un derivado de la misma), ya existente en la librería *BCL*. Esta clase permitirá crear instancias de atributos dinámicamente, al contener toda la metainformación de un atributo cualquiera, pero en esta primitiva se usará para identificar exactamente al atributo requerido. De esta forma, el programador creara una nueva instancia de la clase `FieldInfo`, especificando todas las características deseadas para el atributo a buscar. Esta primitiva reflectiva recibirá esta información, y a partir de la misma buscará el primer atributo que encaje con los datos proporcionados, siguiendo el modelo computacional visto anteriormente.

  Por tanto, este servicio devolverá la información completa de cualquier atributo (sus metadatos), haya sido añadido previamente o no, para su consulta por el programador, de la misma forma que las capacidades de introspección incorporadas en el sistema ya hacen. Además, la operación debe tener en cuenta si el atributo buscado es añadido o no y en caso afirmativo si se ha hecho sobre una instancia o un tipo, ya que se debe tratar la solicitud adecuadamente en función de ello. De acuerdo al modelo computacional visto, una instancia tendrá ese atributo de forma individual mientras que una clase, salvo que sea estático, lo compartirá con todas sus instancias, teniendo que implementarse un mecanismo de evolución de esquemas perezoso en este último caso, como ya se ha descrito. Las figuras 10.1 y 10.2 muestran un diagrama de secuencia y de actividades que ilustran los pasos a seguir por esta operación:

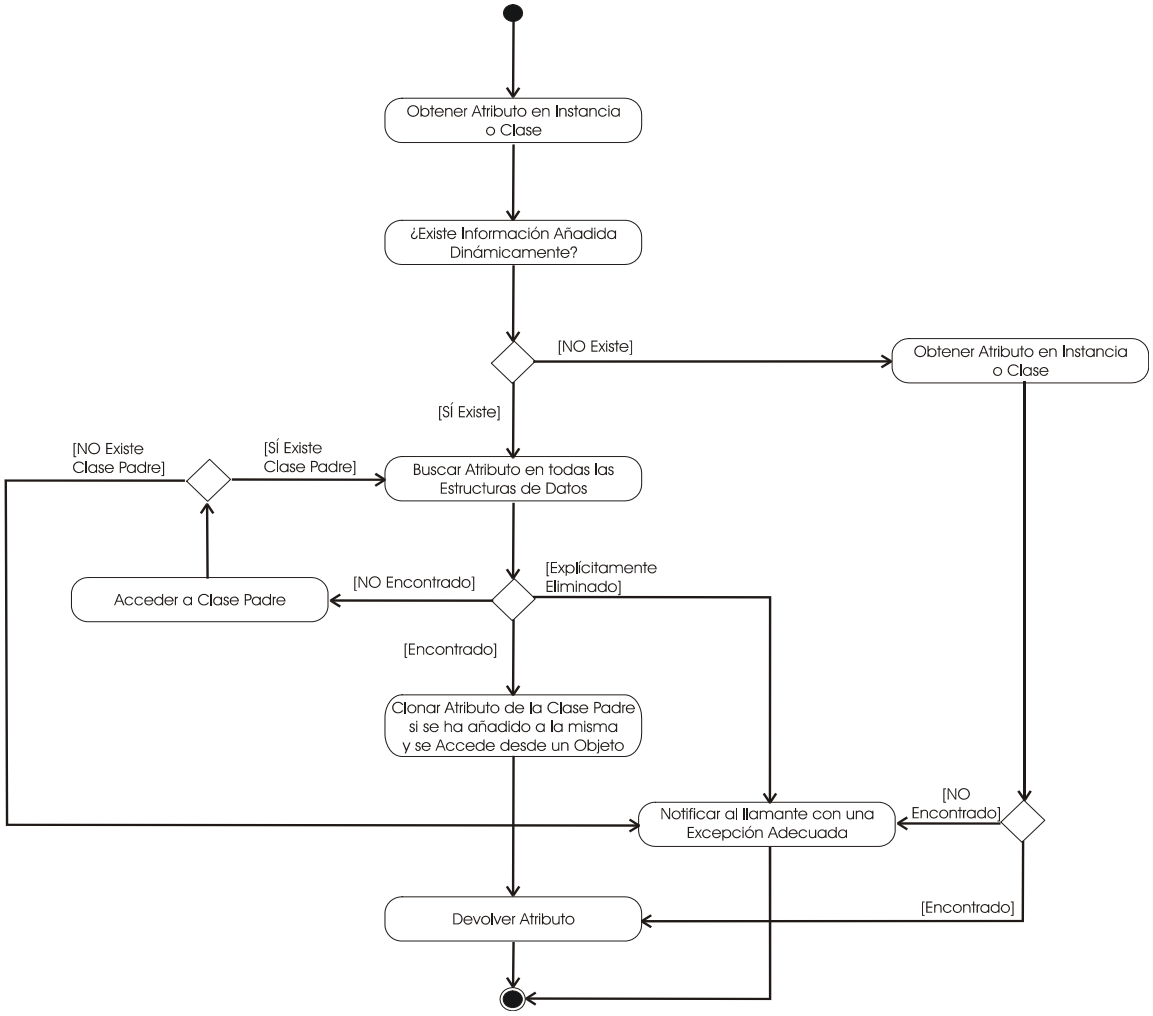

Figura 10.1. Diagrama de actividades del diseño de la primitiva reflectiva "obtener un atributo"

El funcionamiento esperado de la operación consiste en determinar primero si la clase o instancia que se va a manipular en la primitiva reflectiva puede tener o no información añadida dinámicamente, o bien en su jerarquía de clases padre existe alguna que presente información añadida que pueda afectar a la clase o instancia actual. En caso afirmativo, tendremos que realizar una búsqueda en dicha instancia o clase y en toda su jerarquía de clases asociada para localizar el atributo a buscar. Esto permite implementar el modelo de herencia por delegación mencionado en el diseño del modelo computacional, ya que si, al acceder a un atributo, éste no se encuentra en la instancia o clase que se pasa como parámetro, se recorrerá toda su jerarquía para hacer la búsqueda, jerarquía que puede ser alterada dinámicamente por otra primitiva, confirmándose así que ningún elemento tenga una estructura fija e inamovible durante su tiempo de vida. Si no hay información añadida dinámicamente, entonces podremos usar los servicios ya implementados por la máquina para ello (la herencia por concatenación). En cualquier caso, si no se hallase el miembro en cuestión se reportará una excepción que informe del problema.

Por otra parte, en la figura 10.2 podemos ver la interacción entre las distintas entidades involucradas en la ejecución de la primitiva reflectiva. Tal y como se indica en el diagrama de estados, y siendo coherente con el modelo computacional propuesto, al devolver el atributo buscado, si éste ha sido añadido

a una clase y se accede a él por primera vez desde una instancia concreta de esa clase, entonces se deberá devolver un clon de dicho atributo, que se integrará en la instancia en cuestión (mecanismo de evolución de esquemas perezoso) y que permitirá a dicha instancia reflejar los cambios realizados a su clase. Ha de tenerse en cuenta que si el proceso de obtención de atributos detecta información reflectiva añadida en la jerarquía de un elemento cualquiera se efectuará una búsqueda en esta jerarquía, por lo que siempre se podrá localizar un atributo que esté en esta situación.

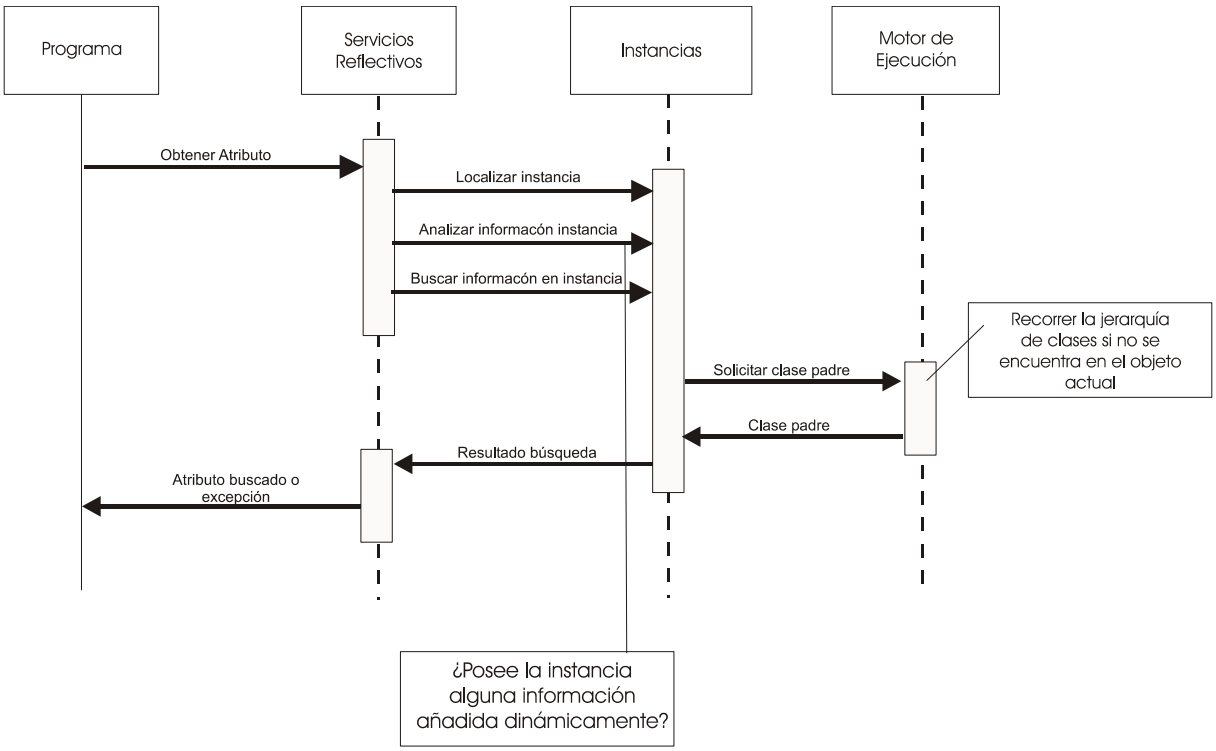

Figura 10.2. Diagrama de secuencia del diseño de la primitiva reflectiva "obtener un atributo"

- **Obtener el valor de un atributo**: Esta operación se puede diseñar de forma idéntica a la anterior, pero sólo devolverá el valor guardado en el atributo buscado. Se ha juzgado necesario incorporar una operación así para hacer más sencilla la obtención de valores únicamente a partir de un atributo dado.
- **Añadir atributo**: Esta operación debe permitir añadir un nuevo atributo a un tipo o a una instancia, poniendo especial énfasis en procurar que no se traten de añadir atributos ya existentes. Según el modelo computacional propuesto, un atributo añadido a un tipo en principio estará accesible para todas las instancias de ese tipo (salvo que sea estático), de manera que automáticamente todas ellas podrán tener una copia privada de la misma. Hemos visto cómo se implementará un mecanismo de evolución de esquemas perezosa a la hora de acceder a dicho atributo, por lo que la operación que permite añadirlos en principio no hará más que simplemente añadirlo al objeto o tipo correspondiente.

  A diferencia de la clase, un atributo añadido a una instancia será sólo individual a la misma, teniendo sólo esa instancia acceso a él, por lo que la operación de añadir atributos deberá alterar únicamente la estructura de esta instancia. Nuevamente para este fin se podrá usar la clase derivada de `FieldInfo` de la que se hablaba anteriormente. Esta clase permitirá crear atributos dinámicamente, al contener toda la metainformación de un atributo a partir de la

cual se podría recrear el nuevo atributo en la zona de memoria destinada a los mismos dentro del motor de ejecución. De esta forma, el programador creara una nueva instancia de la clase `FieldInfo`, especificando todas las características deseadas para el atributo en cuestión. Esta primitiva reflectiva recibirá la información, y a partir de la misma construirá un nuevo atributo integrándolo junto a los ya existentes en la zona de memoria que esté destinada para este fin. Por último, la metainformación de la clase será también situada en la zona de memoria del motor destinada a guardar esta clase de datos, completando así la operación de añadir un atributo guardando por una parte el dato que contiene (información a la que se accederá frecuentemente) y por otra la metainformación asociada a este atributo (información que permitirá obtener datos acerca del atributo en cuestión).

Las figuras 10.3 y 10.4 muestran el diseño de esta operación. En el primero de ellos, si el atributo a añadir no ocasiona ningún conflicto con los ya existentes (nombres repetidos en el mismo ámbito, etc.) entonces se configura adecuadamente (se encapsula de manera idéntica a los atributos ya existentes, como ya se ha mencionado) y se añade. El diagrama 10.4 ilustra la interacción entre los diferentes elementos de la arquitectura para llevar a cabo esta operación.

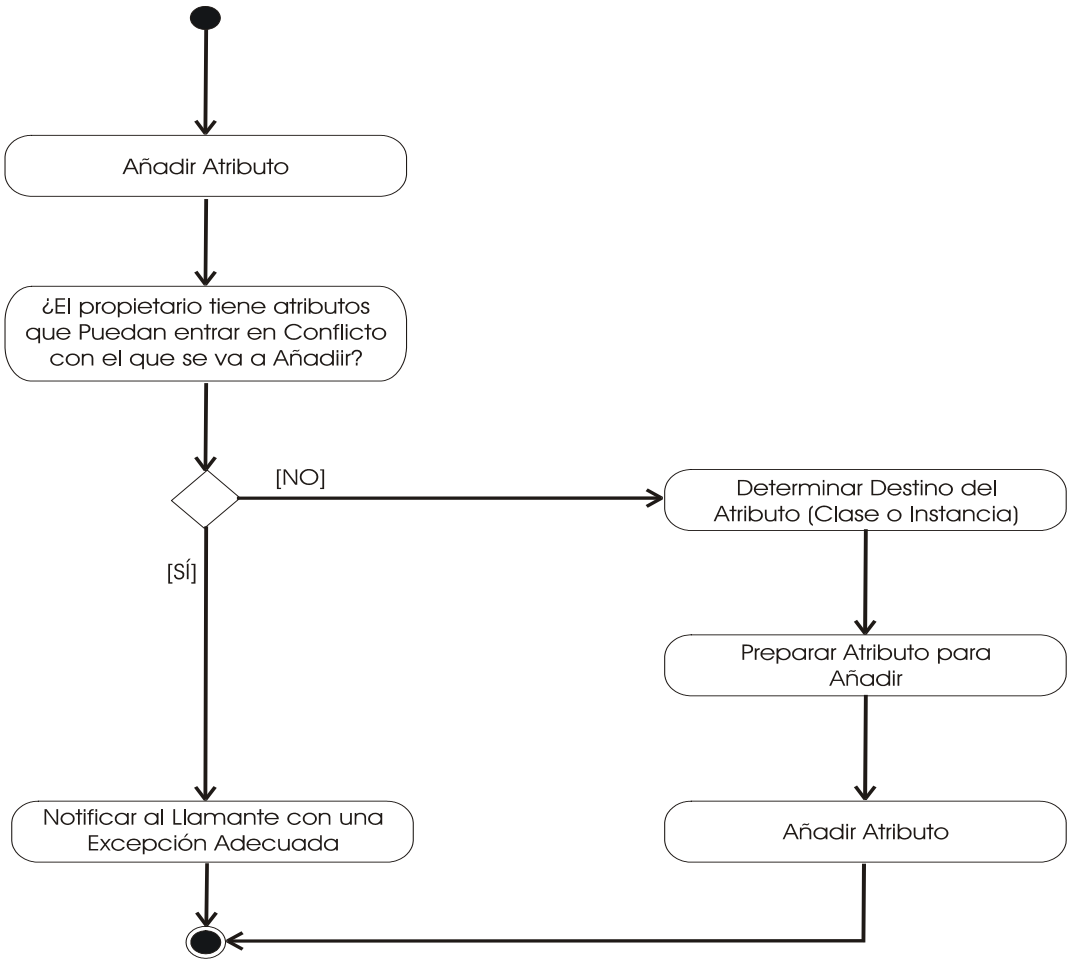

Figura 10.3: Diagrama de actividades de la primitiva reflectiva "añadir un atributo"

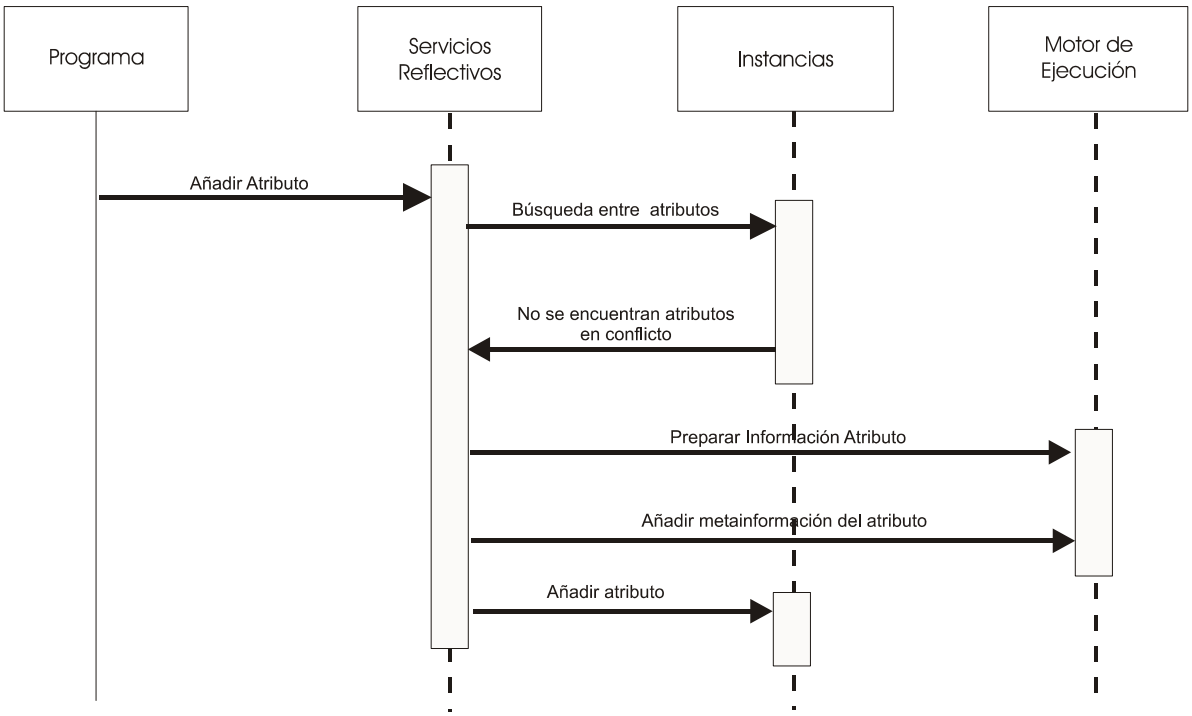

Figura 10.4: Diagrama de secuencia de la primitiva reflectiva "añadir un atributo"

- **Eliminar atributos**: Esta operación deberá hacer inaccesible el atributo especificado para la instancia o tipo que se le proporcione como parámetro a esta operación. La operación deberá tener el mismo comportamiento tanto si el atributo a borrar es añadido dinámicamente como si no: Posteriores accesos al atributo deberán reportar una excepción adecuada para que el usuario pueda tratar cada caso de forma correcta, como en el caso de la operación de añadir atributos. El proceso llevado a cabo por esta operación es similar al mecanismo de delegación mostrado en la figura 10.2 (obtener atributo) sólo que en este caso el atributo se elimina, no se devuelve. La principal dificultad existente en esta operación es cómo efectuar la eliminación física de los atributos. Se debe tener especial cuidado a la hora de hacer dicha eliminación, puesto que aunque en un principio bastaría con eliminar de memoria toda referencia a un atributo, y dejar que el mecanismo de búsqueda ya descrito se encargue de notificar el error si se trata de acceder a el nuevamente, es posible que la disposición de la información dentro de la implementación escogida, o bien los convenios usados por su motor de ejecución entre otros motivos puedan hacer aconsejable "marcar" la información como eliminada antes de eliminarla físicamente. De todas formas, éste es un aspecto de implementación que abordaremos más adelante.
- **Modificar atributos**: Esta operación deberá sustituir un atributo por otro a partir de su nombre. Permitiría alterar todas las características (tipo, visibilidad, valor,...) del atributo y el efecto conseguido puede ser equivalente a eliminar el atributo antiguo y añadir uno nuevo, por lo que no se profundizará más en su estudio.
- **Fijar el valor de un atributo**: Con esta operación simplemente se fijará el valor de un atributo dado, sea añadido o no. La operación deberá obtener el atributo (empleando la primera primitiva descrita) y luego proceder a cambiar su valor.
- **Existencia de un atributo**: Esta operación deberá determinar si un tipo o instancia tiene un determinado atributo cuyo nombre se le proporcione, tanto si es añadido dinámicamente o no. El efecto a conseguir sería equivalente a la operación de obtención de un atributo, sólo que en este caso deberá devolverse si existe o

no en lugar del atributo en sí, de manera que este tipo de consultas puedan hacerse más eficientemente.

10.3.2 Operaciones sobre Métodos

Al igual que en el apartado anterior, se describirán a continuación las operaciones a realizar sobre los métodos de una clase o instancia:

- **Obtener un método**: A partir de un tipo o instancia, e información acerca de la signatura de un método, este servicio deberá obtener la información relativa a un método sin que, por supuesto, se haga distinción patente entre métodos añadidos dinámicamente e introducidos por código la hora de obtenerlos, al igual que se hizo con los atributos. También debe contemplar casos como la inexistencia del método pedido, o la no concordancia de elementos de la signatura proporcionada con los métodos existentes, en cuyo caso deberá devolver la excepción correspondiente. Al igual que en el caso de los atributos se deberá crear una representación de los métodos que permita tratarlos homogéneamente independientemente de su origen, en este caso derivada de la clase `MethodInfo` ya existente. Debe también tenerse en cuenta si el método pertenece a un objeto (individual de ese objeto) o a una clase (*trait object*, compartido por sus instancias, que en el modelo de prototipos serían objetos relacionados con ese *trait object*) para completar correctamente las operaciones pedidas. Debemos considerar también que en este caso los métodos de las clases seguirán siendo propiedad de las mismas, y no hará falta hacer una evolución de esquemas a diferencia del caso de los atributos.

  El mecanismo de herencia basado en delegación que usan los lenguajes dinámicos, cuyo funcionamiento previsto hemos expuesto en la operación de localización de un atributo, será aplicable tal y como se ha descrito en dicha operación anterior a la hora de localizar un método. En líneas generales, tanto el diagrama de estados como de secuencia presentado para el caso de los atributos describen también la obtención de un método, usando su signatura como clave de búsqueda.

- **Añadir un método**: Este servicio debe añadir el método especificado al tipo o instancia que se le indique, siendo dicho método utilizable posteriormente el mismo de igual forma que si hubiese sido declarado estáticamente en el código del programa. El funcionamiento esperado es similar al de los atributos: Si el destinatario es un tipo (*trait object*), entonces todas las instancias de dicho tipo deberán poder usar este nuevo método. Si por el contrario es una instancia, entonces sólo dicha instancia podrá emplear este método. Además, debe tenerse especial cuidado con métodos cuya signatura sea distinta a alguno ya existente, ya que se debe permitir la sobrecarga de métodos.

  El principal problema que plantea esta operación es crear una entidad que permita guardar toda la información de un nuevo método, de manera que dicha información pueda representarlo hasta el punto de poder invocarlo, lo que implicaría tener acceso a su código completo. En la plataforma seleccionada, todo método existente en el sistema pertenecerá a una clase concreta C. Esto quiere decir que si se pretende emplear un método M, perteneciente a C, para añadirlo a una clase diferente C2, debemos tener en cuenta dicha pertenencia a C como

punto de partida para esta operación[12]. El hecho de pertenecer a una clase C significa, entre otras cosas, que todo acceso a atributos o métodos hecho de manera explícita o implícita al objeto `this` dentro del código de ese método se contrastará con la estructura de C para su ejecución. Por tanto, si se pretendiese usar M sobre otra clase C2, el usuario deberá ser consciente de que C2 debe ofrecer los atributos o métodos que M emplee también, bien en una forma idéntica o compatible con los originales de C, para que de ese modo la ejecución de M sobre C2 sea completamente coherente y no carezca de sentido. Teniendo en cuenta esto último, existen dos posibles formas viables de lograr la ejecución de un método M cualquiera sobre otra clase distinta a la que lo declaró C2:

- Alterando el tipo de la referencia `this`: Si se logra alterar el parámetro implícito `this` para que, en vez de contener un elemento de tipo C, contenga uno de tipo C2, si C2 tiene la información necesaria para que el método pueda completarse correctamente, no debería ocurrir ningún problema para ejecutar M sobre C2. Esta operación es viable alterando las comprobaciones que el motor de ejecución haga al efecto si se detecta que el método que se va a usar ha sido añadido a la clase. La ventaja principal de esta operación es que permitiría "reutilizar" cualquier método ya creado para ser empleado sobre un objeto cualquiera que sobre el que usuario desee aplicarlo.
- Crear un nuevo método a partir del original: Esta técnica consiste en crear, mediante la funcionalidad proporcionada por el paquete `System.Reflection.Emit`, un nuevo tipo C' que posea a su vez un nuevo método M', construido a partir del M original. Este método M' será una copia casi exacta del M original, con la salvedad que recibirá un parámetro extra (el objeto de tipo C2) y todas sus referencias a `this` serán redirigidas a este primer parámetro. Esta posibilidad es viable actualmente dadas las posibilidades de las librerías del *CLI*, pero tiene un elevado coste de ejecución (el método M puede ser muy complejo y la operación de copia puede ser muy costosa) y produce fragmentación (cada método añadido conlleva la creación de un tipo nuevo, lo que por otra parte eleva el coste de la operación).

La primera de las opciones será la escogida para crear nuevos métodos, por lo que todo método que se vaya a usar para ser incorporado a una clase cualquiera deberá existir previamente en el programa. La ventaja principal consiste en que la información del método que se va a añadir (sus metadatos) ya estará creada (crearla dinámicamente nosotros mismos puede ser excesivamente complicado, sobre todo teniendo en cuenta aspectos como convenios de llamada, código ejecutable, etc.), teniendo sólo que guardar información acerca de que el método M ahora es aplicable sobre la clase C2 además de sobre la clase C. Debe tenerse en cuenta que M nunca se duplica, simplemente se amplia el espectro de clases que pueden invocarlo. Esto no supone una pérdida de funcionalidad, puesto que el *CLI* ya contempla la posibilidad de crear métodos dinámicamente usando el mencionado paquete `Emit`, por lo que un programa podría generar un nuevo método dinámicamente y añadirlo a una clase cualquiera. El diagrama 10.5 muestra el comportamiento deseado por esta funcionalidad. Nótese que, siguiendo lo explicado anteriormente, al añadir el método creado se supone que se ha elaborado uno a partir de la información del existente que se pretende añadir

12 Esto no quiere decir que todo metodo M que se quiera añadir a una clase C2 tenga que estar ya creado estáticamente, ya que tambien es posible que M se cree dinámicamente usando las capacidadades del paquete System.Reflection.Emit

(signatura, tipo de retorno,...) y que esta copia apuntará al código del original.

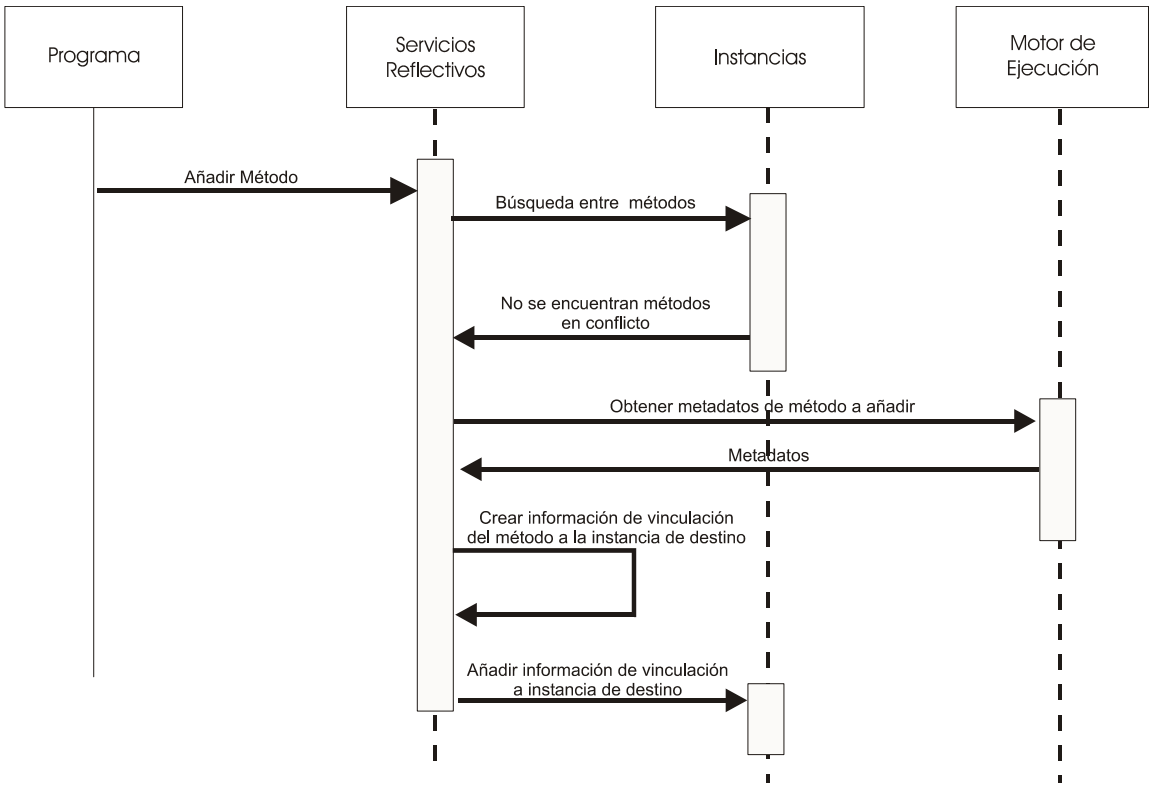

Figura 10.5: Diagrama de secuencia de la primitiva reflectiva "añadir un método"

- **Eliminar método**: A partir de un tipo o instancia y una signatura de un método, el servicio deberá hacer que este método ya no esté disponible para ser llamado en un futuro, sin importar el origen dinámico o no del mismo. De igual modo, debe identificar posibles incompatibilidades y problemas detectados y notificarlos vía excepciones, para poder reaccionar adecuadamente ante estos casos. Un método eliminado sobre una clase dejará de estar disponible para todas sus instancias, mientras que si se elimina sobre una instancia, sólo ella misma se verá afectada por los cambios. Esta operación planeta además un problema de sincronismo, puesto que en un entorno en el que se manejen varios hilos de ejecución puede ocurrir que se elimine un método que en ese preciso instante esté en ejecución. Si eso ocurriese en principio tenemos dos posibles opciones:

    - Continuar normalmente con la ejecución del mismo: El método se seguiría ejecutando, pero no podría volver a llamarse una próxima vez ya que entonces se detectaría su eliminación.
    - Abortar inmediatamente la ejecución del mismo: El método se interrumpe y se devuelve una excepción indicando que el método ya no existe.

En nuestro caso hemos considerado que era más coherente con el modelo computacional implementado, y no planteaba ningún inconveniente que considerásemos relevante, implementar la primera opción. De esta forma una vez que el método se encuentre en la búsqueda y se haga la llamada al mismo, este método se ejecutará hasta su finalización, dándose cuenta el sistema de su

eliminación la próxima vez que se intente hacer acceso al mismo. Además de la sencillez, otra de las ventajas de esta aproximación es que no cortaríamos ninguna operación que estuviese en plena realización, caso en el que podríamos dejar al sistema en un estado potencialmente inconsistente.

- **Alterar método**: El propósito de este método es modificar las características de un método dado, lo que a efectos de uso equivaldrá a modificar la implementación del método cuya signatura se le proporcionará. Esta operación puede considerarse un equivalente a eliminar el método anterior y añadir uno nuevo, por lo que no se profundizará más allá en la descripción de la misma.
- **Existencia de métodos**: Operación para buscar si un tipo o instancia posee un determinado método cuya signatura se le proporciona, bien sea añadido dinámicamente o no, al igual que la operación equivalente con atributos.
- **Invocación**: Este servicio se encargará de invocar a cualquier método existente en cualquier tipo o clase, a partir de su signatura y valores de los parámetros que se le suministren, de una forma análoga al `invoke` que posee la clase `MethodInfo` nativa del sistema, pero contemplando adicionalmente los métodos añadidos. El comportamiento en caso de errores deberá ser el esperado por otras operaciones ya descritas, lanzando excepciones en dichos casos. La figura 10.6 ilustra cómo será la operación indicada, teniendo en cuenta las consideraciones ya expuestas a la hora de añadir un método.

  Debe tenerse en cuenta que esta misma semántica de ejecución la poseerá tanto el método que permita una invocación dinámica explícita desde una clase del *API* de reflexión implementado dentro de la máquina como la instrucciones del *CIL* `call` y `callvirt`, que se ocupan de la invocación de métodos, por lo que esta funcionalidad se puede lograr de forma transparente y no sólo mediante la llamada explícita al servicio mencionado.

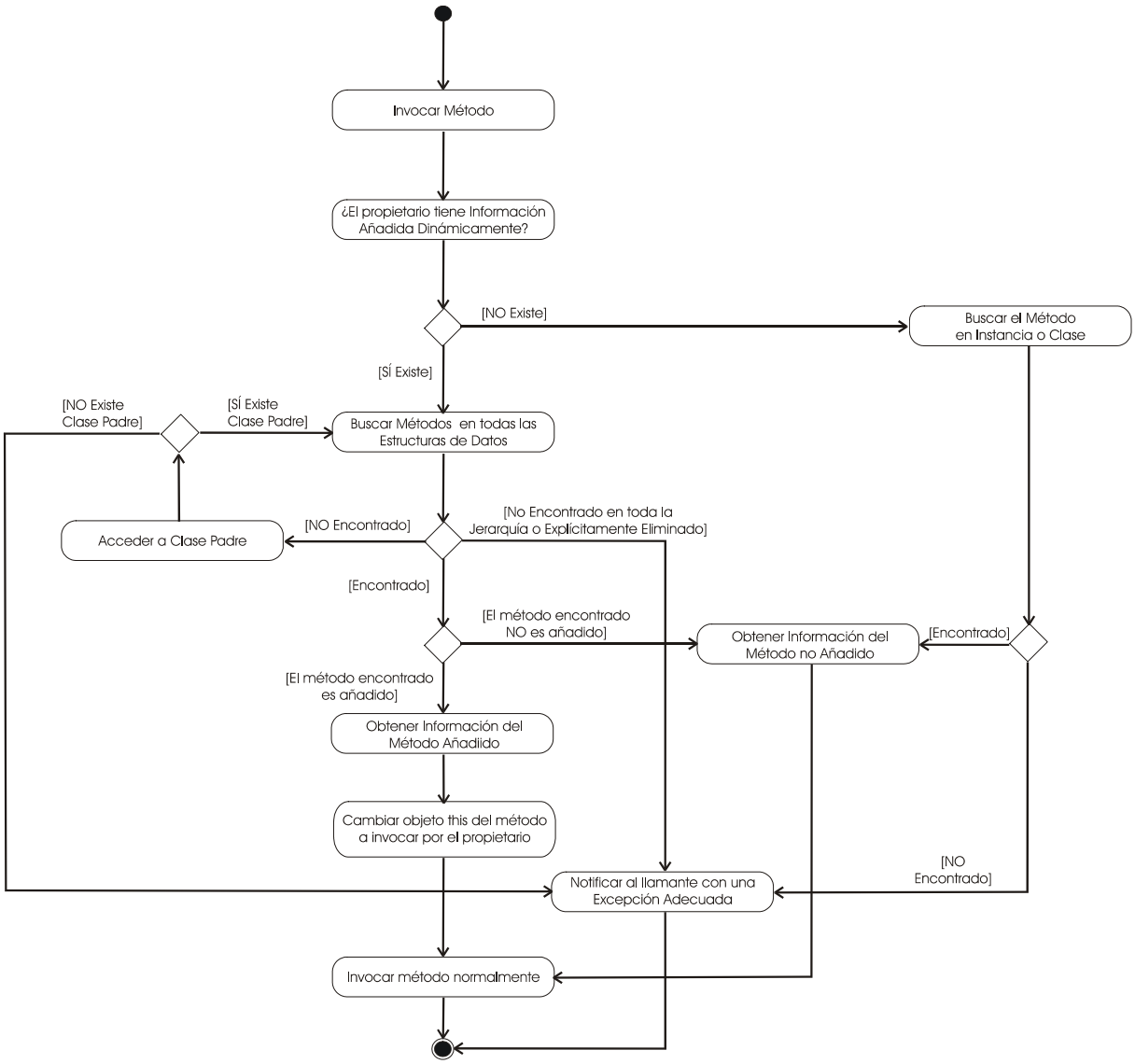

Figura 10.6: Diagrama de actividades de la primitiva "invocar un método"

11 INTEROPERABILIDAD DE LENGUAJES

11.1 SOPORTE PARA INTEROPERABILIDAD

Hasta ahora hemos visto cómo el estándar *CLI* especifica un modelo de objetos orientado a clases, que sólo le permite ejecutar eficientemente lenguajes estáticos, y cómo, para ejecutar lenguajes dinámicos coherentemente, hemos ampliado ese modelo computacional usando el modelo de orientación a objetos basado en prototipos, conservando todas las características del sistema para mantener la compatibilidad, de forma que ambos tipos de lenguajes puedan ser ejecutados sobre la máquina adecuadamente. Estas modificaciones también deben influir sobre la interoperabilidad que el sistema oferta: cualquier lenguaje dinámico que se pueda ejecutar sobre la plataforma debe poder interoperar con cualquier lenguaje (dinámico o no) ya existente.

Por tanto, la máquina virtual *CLI* extendida será ahora una plataforma de bajo nivel que ejecutará un amplio conjunto de lenguajes de alto nivel, y que soportará tanto el modelo de orientación a objetos basado en clases como el modelo de orientación a objetos basado en prototipos, permitiendo el primero ejecutar aplicaciones estáticas basadas en clases y el segundo aplicaciones dinámicas y reflectivas basadas en prototipos. Es necesario pues algún elemento discriminador que permita indicar qué funcionalidades del motor del sistema van a usar cada uno de los lenguajes, de manera que las operaciones llevadas al cabo por el motor respondan a las necesidades de cada lenguaje de alto nivel correctamente. Este elemento discriminador son los compiladores de los lenguajes implementados sobre la plataforma, que actuarán como "filtros" seleccionando los servicios del modelo apropiado dependiendo del lenguaje que se compile y su carácter. Esto se ilustra en la figura 11.1.

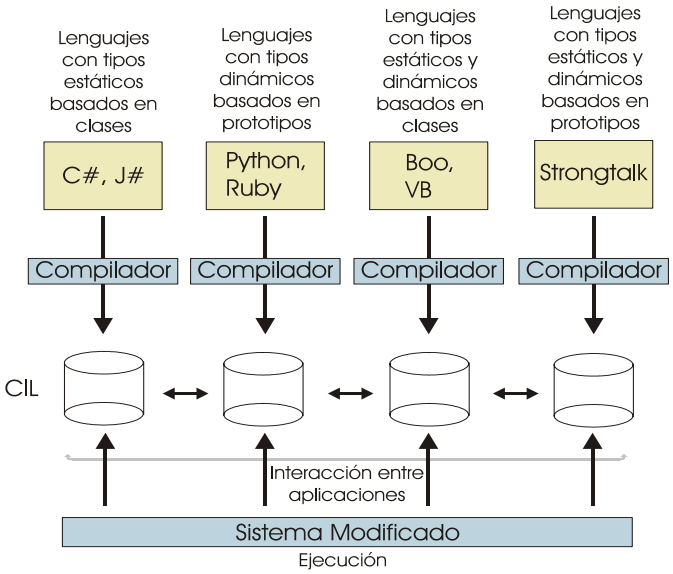

Figura 11.1 Soporte para los modelos computacionales de diferentes lenguajes orientados a objetos

Vemos cómo la máquina podrá soportar ahora un amplio abanico de lenguajes de diferente naturaleza. Podemos establecer las siguientes consideraciones al respecto:

- **Lenguajes con tipos estáticos basados en clases**: En nuestra versión extendida del sistema debe ser posible ejecutar cualquier lenguaje previamente existente para la plataforma *CLI* original, y serán estos programas precisamente los que usen el modelo de objetos orientado a clases original que poseía la plataforma, sin usar ninguna capacidad reflectiva de las introducidas en el sistema.
- **Lenguajes con tipos dinámicos basados en prototipos**: Para soportar lenguajes dinámicos que permitan reflexión estructural (como *Python* o *Ruby*), se añaden las primitivas reflectivas vistas, con la semántica del modelo de prototipos que ya ha sido analizada. En consecuencia, cualquier compilador de un lenguaje dinámico será capaz de usar directamente estos nuevos servicios de la plataforma. No hay necesidad de generar código adicional para simular el modelo reflectivo sobre el estático existente.

  Para soportar el modelo basado en prototipos que usan estos lenguajes, las clases existentes en la plataforma se podrán comportar pues como *trait objects*. El compilador usará de esta forma los nuevos servicios de la máquina extendida para permitir modificar el comportamiento introducido en las clases, pero no su estructura, para tener así el mismo funcionamiento de los *trait objects* definidos anteriormente. Para el caso de los objetos, el compilador podrá también usar los servicios del sistema para alterar tanto su estructura como su comportamiento de manera que se soporten cambios en los mismos sin que se produzcan conflictos con sus clases, que al comportarse como *trait objects*, ya no están vinculadas estructuralmente a las instancias.

  Por otra parte, los lenguajes dinámicos de esta categoría hacen uso de invocaciones dinámicas por delegación, servicio que ahora está soportado nativamente en el sistema a través de las instrucciones `call` y `callvirt` del *CIL* de la máquina (dada la ampliación de su semántica) y que por tanto podrán usar de forma transparente. Esta clase de lenguajes dinámicos también requieren de inferencia dinámica de tipos, ya que en ellos no es necesario declarar el tipo de las variables, servicio que también se encuentra disponible en el sistema ya que, tal y como se ha descrito anteriormente, este tipo de comprobaciones han sido pospuestas en el compilador *JIT* hasta el ultimo momento posible (en tiempo de ejecución).
- **Lenguajes con tipos estáticos y dinámicos basados en clases**: Como se ha visto en el estudio de sistemas existentes, existen además ciertos lenguajes de programación destinados a la plataforma *.NET* que usan tanto tipos estáticos como dinámicos (permiten tipos estáticos por motivos de eficiencia), como *Visual Basic .NET* o *Boo* [Boo05] (*duck typing*). Estos lenguajes no soportan reflexión estructural y su sistema de tipos dinámicos sólo está basado en la introspección.

  Las características dinámicas de estos lenguajes se beneficiarán del mejor rendimiento de la máquina virtual extendida y son también soportados por ella, ya que hemos visto cómo una de las nuevas capacidades introducidas es un sistema de inferencia dinámica de tipos que les permite ser implementados directamente sobre la máquina, usando este servicio para soportar los tipos dinámicos que estos lenguajes poseen. En cuanto al uso de tipos estáticos, los lenguajes se aprovecharán del soporte ya existente para ellos en el sistema original. Por último, conviene recordar que estos lenguajes están basados en clases (figura 11.1) y por tanto seguirán usando el modelo de orientación a objetos basado en clases que el sistema original ya poseía.
- **Lenguajes con tipos estáticos y dinámicos basados en prototipos**: Existen

además otro tipo de lenguajes donde el uso de tipos estáticos es algo opcional, introduciendo el chequeo de tipos en un lenguaje dinámico que usa modelo de objetos basado en prototipos sin afectar a la flexibilidad, puesto que éste se realizará a petición del programador. Un lenguaje que usa esta aproximación es *Strongtalk*, una modificación de *Smalltalk* [Bracha93]. Un compilador de este lenguaje puede inferir tipos estáticamente o posponer esta operación a tiempo de ejecución, usando al mismo tiempo ambas capacidades de la máquina virtual extendida y el soporte para lenguajes cuyo modelo de objetos esté basado en prototipos que ya se han descrito en los puntos anteriores.

Será pues el compilador de cada uno de estos lenguajes el que decida qué servicios disponibles en la máquina extendida quiere usar y cómo, proporcionando el sistema extendido soporte eficiente para todos ellos. En la explicación dada anteriormente se ha visto también cómo, entre todos los tipos de lenguajes presentados, se han usado ambos modelos de objetos y todas las nuevas capacidades introducidas en la máquina para soportarlos adecuadamente.

11.2 MODELO DE INTERACCIÓN ENTRE LENGUAJES

Teniendo en cuenta todo lo mencionado acerca de los lenguajes soportados y las modificaciones realizadas al sistema, haremos ahora una breve descripción de qué comportamiento tendrá el sistema si se requiere interoperabilidad entre los diferentes lenguajes estáticos y dinámicos soportados por la máquina, es decir, cómo va a ser posible acceder a las entidades de un modelo desde el punto de vista del otro. En este punto sólo se van a proponer una serie de ideas tentativas acerca de cómo se podría interactuar entre ambos modelos para demostrar que realmente es posible, ya que un desarrollo más detallado de este aspecto sería una vía de investigación futura y está fuera del alcance de esta tesis:

11.2.1 Acceso de un Lenguaje OO basado en Prototipos a un Lenguaje OO basado en clases

En este apartado vamos a tratar de ver cómo un lenguaje orientado a objetos basado en prototipos accederá a entidades (clases y objetos) que han sido definidas en un lenguaje orientado a objetos basado en clases. Previamente a explicar el esquema que permita a ambos interoperar, se deben hacer dos consideraciones importantes:

- El uso de un modelo de prototipos no conlleva una pérdida de semántica respecto al uso del modelo de clases, es decir, todo aquello modelable mediante un modelo de clases es modelable también por un modelo de prototipos, no existe pérdida de expresividad al hacer la transición entre modelos [Ungar91] [Evins94].

- El modelo de prototipos es más flexible que el modelo de clases, ya que a lo largo de esta tesis hemos visto como este último modelo tiene problemas para soportar coherentemente determinadas operaciones flexibles.

Una vez visto esto, debemos considerar pues qué entidades va a poder ver un modelo de prototipos de un modelo de clases. Un modelo de clases posee dos entidades diferentes: las clases y las instancias de las clases (objetos). Para un lenguaje basado en prototipos una clase va a ser considerada un objeto *trait*. Por tanto, los métodos existentes en esa clase serán los de dicho objeto *trait* y la asociación del comportamiento entre ambos elementos de los distintos modelos se hace directamente. La semántica no varia, a pesar de que en ambos casos la implementación es diferente (un lenguaje basado en clases posee un modelo de herencia por concatenación frente al modelo de herencia por delegación de un lenguaje basado en prototipos). Como hemos visto, un objeto *trait* no posee información de estado y por ello la traducción se hará de manera que el objeto *trait* sólo poseerá comportamiento y atributos estáticos (miembros compartidos).

Por otra parte, las instancias de un modelo basado en clases son las encargadas de guardar el estado particular de cada objeto, y de esta forma serán traducidas al modelo de prototipos, como objetos de este modelo. La asociación estado - comportamiento existente en una clase quedará así pues dividida en dos entidades separadas en el modelo de prototipos (objeto y *trait object* asociado a ese objeto), de forma que se traduzcan a conceptos existentes en dicho modelo.

Debe tenerse en cuenta que en el modelo de prototipos no existe el concepto de clase, y por tanto el conjunto de atributos de un objeto no puede estar determinado por ninguna entidad externa. Por ello, la forma que tendría el modelo de prototipos de acceder al conjunto de atributos (estado) de un objeto sería mediante introspección, obteniendo así la lista de atributos directamente desde cada objeto y no dejando a la clase esta responsabilidad. La siguiente imagen muestra un ejemplo de esta interacción, donde vemos cómo, manteniendo la misma identidad, las entidades son representadas en cada uno de los modelos de objetos:

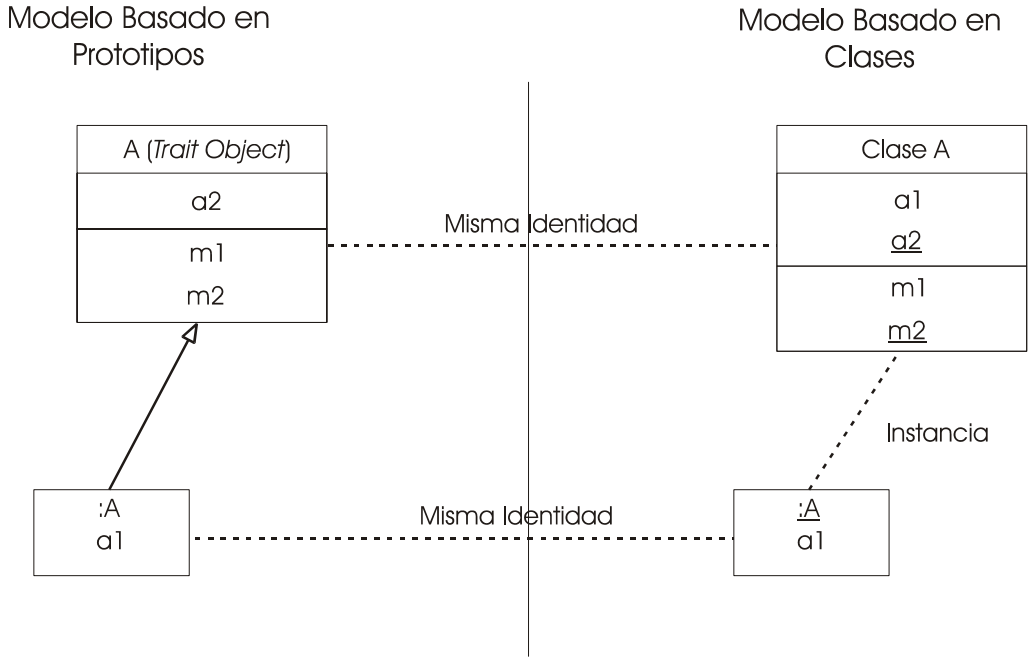

Imagen 11.2: Modelo de prototipos interactuando con un modelo de clases

Debe tenerse en cuenta que en todo momento estamos tratando de mantener la identidad, es decir, que desde el modelo de prototipos se acceda a elementos del modelo de clases y que en ambos modelos sean la misma entidad, de manera que los cambios de estado en un modelo sean visibles desde el otro. Si quisiésemos crear un nuevo objeto en el modelo de prototipos a partir de una clase (del *trait object* por el que estaría representado en el modelo de prototipos), entonces podríamos hacer uso de introspección y reflexión estructural para que, desde el código que haga de constructor de los objetos, se puedan añadir (mediante reflexión estructural) todos los atributos de instancia que la clase posea (leídos mediante introspección) en cada instancia creada. De esta manera cada objeto tendrá una estructura particular (que nuevamente podrá ser leída mediante introspección) y no se hace que la clase determine en todo momento la misma.

11.2.2 Acceso de un Lenguaje OO basado en Clases a un Lenguaje OO basado en Prototipos

En este caso se trataría de hacer que un lenguaje orientado a objetos basado en clases pudiese acceder a las entidades definidas en un lenguaje orientado a objetos basado en prototipos, de manera que puedan acceder a estado y comportamiento definido en las entidades del mismo e interactuar con éstas coherentemente. En este caso el problema al que nos enfrentamos es cómo representar en un modelo basado en clases toda la flexibilidad de la que pueden disponer las entidades del modelo basado en prototipos. A continuación vamos a exponer tres escenarios diferentes en los que puede querer hacerse la interacción entre modelos.

11.2.2.1 INTEROPERABILIDAD CONSERVANDO LA IDENTIDAD DEL OBJETO

A lo largo de esta tesis se han visto las diferencias existentes entre ambos modelos de objetos en cuanto a las entidades que son capaces de guardar estado y comportamiento. El problema existente para que ambos modelos interactúen es cómo hacer que el modelo de objetos basado en clases sea capaz de conservar la identidad de los objetos existentes en el modelo de prototipos, permitiendo llamar a todos los métodos y acceder a todos los atributos que definan éstos, a pesar de que ahora guardan una organización diferente.

Un problema aún más importante es cómo reflejar en el modelo de clases el resultado de las operaciones de reflexión estructural que se pueden producir sobre el modelo de prototipos, ya que un modelo de clases no soporta reflexión estructural y las modificaciones que se hagan a los objetos pueden por tanto no ser correctamente modeladas en dicho modelo de clases. Además, en este caso es muy importante tener en cuenta estos cambios, ya que queremos conservar la identidad de los objetos entre modelos, y por tanto cualquier cambio hecho sobre un objeto en el modelo de prototipos debe ser tenido en cuenta en la "vista" de ese objeto existente en el modelo de clases.

Una posible solución para poder llevar a cabo la interoperabilidad entre modelos en este caso es crear un delegado que, mediante introspección, permita acceder tanto a atributos como a métodos de un objeto asociado en el modelo de prototipos, de manera que sea capaz de leer miembros de dicho objeto o de los objetos *trait* que tenga

asociados, para así dar al modelo de clases una entidad conceptualmente similar a un objeto estándar de este modelo, que conserva estado particular y permite acceder a todos los métodos asociados con el mismo. La entidad intermedia podría tener las siguientes operaciones reflectivas para este fin:

- ***Invoke***: Invocaría al método especificado con los parámetros adecuados, buscando dónde está definido en el modelo de prototipos (el propio objeto o un *trait* asociado) e invocándolo con los parámetros suministrados.
- ***Get/SetValue***: Accedería/modificaría un valor de un atributo de ese objeto.

El esquema de cómo resultaría la estructura creada aparece en la siguiente imagen, donde se ve cómo, a partir de una clase que actuaría como *proxy,* se podría acceder dinámicamente a la información representada en el modelo de prototipos, abstrayéndose de cómo esté organizada:

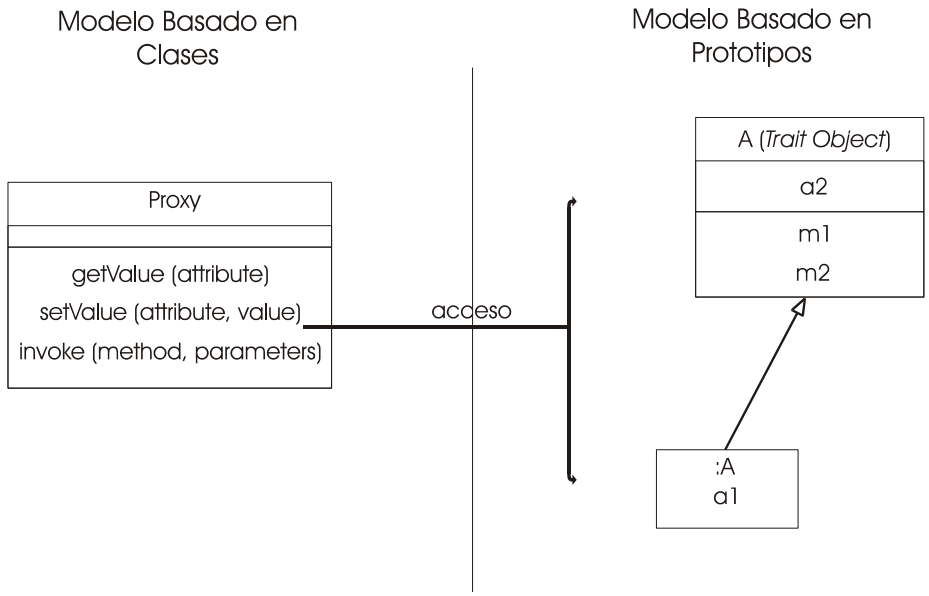

Imagen 11.3: Interoperabilidad clases - prototipos manteniendo la identidad

Esta aproximación es válida para el propósito indicado, pero tiene un inconveniente: la detección de errores de tipos en tiempo de compilación. Cuando se usa un lenguaje basado en clases con tipos estáticos, se espera que el compilador sea capaz de decir si el uso de los tipos es válido o no antes de que el programa se ejecute, para eliminar posibles errores en tiempo de ejecución. Con este modelo esto ya no va a ser posible, debido a que las llamadas vía introspección harán comprobaciones de manera dinámica y no será posible saber si un atributo al que queremos acceder o un método al que queremos invocar va a existir en todo momento que se requiera acceso al mismo. En caso de que un atributo o método no exista, habría que lanzar una excepción que el programa podría capturar y tratar para actuar en consecuencia, pero se estría perdiendo la comprobación estática de tipos.

No obstante, si queremos conservar la identidad del objeto en ambos modelos, ésta es una solución válida. En caso de que no nos interese conservar la identidad, entonces se pueden seguir otras aproximaciones que sí permitan el chequeo estático de tipos y que veremos en los dos siguientes apartados.

11.2.2.2 ACCESO AL ESTADO DEL OBJETO

En este caso el modelo de clases va a interaccionar con el de prototipos, pero no requerirá conservar la identidad del objeto con el que interactúa para su funcionamiento. Esta parte está dividida en dos subcasos, pero en ambos hay un denominador común: se creará una clase que modele el estado y comportamiento de las entidades con las que se va a trabajar, a partir del contenido de éstas en un momento concreto de la ejecución, de manera que esta clase o clases representan a los objetos del modelo de prototipos en un momento dado (algo que podemos hacer puesto que, como hemos dicho, ya no nos interesa conservar la identidad de los objetos, y por ello cualquier cambio que se haga en los mismos ya no requiere trasladarlo a la "vista" del objeto en el modelo de clases).

ESCENARIO 1: TRASLADAR ESTADO Y COMPORTAMIENTO AL MODELO DE CLASES

En el modelo basado en prototipos, el comportamiento de un objeto puede estar contenido en un objeto *trait* y también dentro de la estructura del propio objeto. Supongamos que en el primer caso que vamos a estudiar el objeto no posee comportamiento propio. En este caso, se debe generar una clase que contenga los métodos de los *trait objects* asociados al objeto y los atributos del propio objeto.

Para modelar los atributos, debemos crear una estructura de datos que los contenga y que se pueda rellenar dinámicamente a la hora de crear una nueva instancia (ya que, para modelar adecuadamente el modelo de prototipos, no podemos hacer una declaración estática de los mismos, que produciría una clase distinta para cada combinación de atributos existente). Esto se puede hacer añadiendo a dicha clase un atributo que sea una estructura de datos tipo contenedor asociativo, que permita guardar pares <clave, valor>, donde la clave sea el nombre del atributo y el valor sea el valor del mismo en un momento dado. Esto permitiría tener un contenedor con tantos elementos como atributos tenga el objeto de partida.

En el caso de los métodos, tendremos que copiar el conjunto de métodos (comportamiento) asociados al objeto del que se parte en la clase creada. Esto deberá hacerse mediante `System.Reflection.Emit`, asegurándonos antes de que el método sea válido para el modelo de clases (es decir, que no pretenda alterar la estructura y el comportamiento de las clases si el lenguaje basado en clases no lo permite). Este proceso de copia sería una traslación literal de cada *opcode* del método a un nuevo método, pero existe una excepción al respecto: los *opcodes* de acceso a atributos (como `ldfld, stfld`, etc.) deberán ser cambiados por un código que convierta estos accesos a atributos a una consulta al contenedor asociativo de la nueva clase por el nombre del mismo, para de esta manera lograr la misma funcionalidad que el método ya poseía.

Por último, si el objeto del que se parte tiene comportamiento particular, se puede mantener la estructura presentada íntegramente, pero será necesario crear una subclase de la vista anteriormente que, siguiendo el mismo procedimiento explicado, añada los métodos que son propios del objeto. De esta forma, se consigue trasladar los objetos definidos en el modelo de prototipos al modelo de clases, conservando tanto comportamiento como estado en entidades que son utilizables por el modelo de clases y guardando una total coherencia con el contenido de los objetos de partida.

En la siguiente imagen se ve cómo se va a realizar esta interacción entre modelos, donde los objetos que estarían representados por cada uno de los objetos *prototype* existentes en el modelo de prototipos y su objeto *trait* asociado son convertidos en dos clases, que contengan los métodos del *trait object* y una tabla *hash,* para guardar tanto nombre como valor de aquellos atributos que posea cada uno de los prototipos. En las instancias de estas clases, que representarían a los objetos del modelo de prototipos, se rellenará esta tabla *hash* con los atributos y valores correspondientes. Por último, nótese

cómo la creación de la segunda clase, derivada de la primera, es necesaria para modelar el comportamiento particular poseído por uno de los objetos prototipo, de manera que en el modelo de clases sólo los objetos que sean instancias de esta segunda clase posean este comportamiento, y creando de esta forma la estructura que siga más fielmente a la que estaba presente en el modelo de prototipos original.

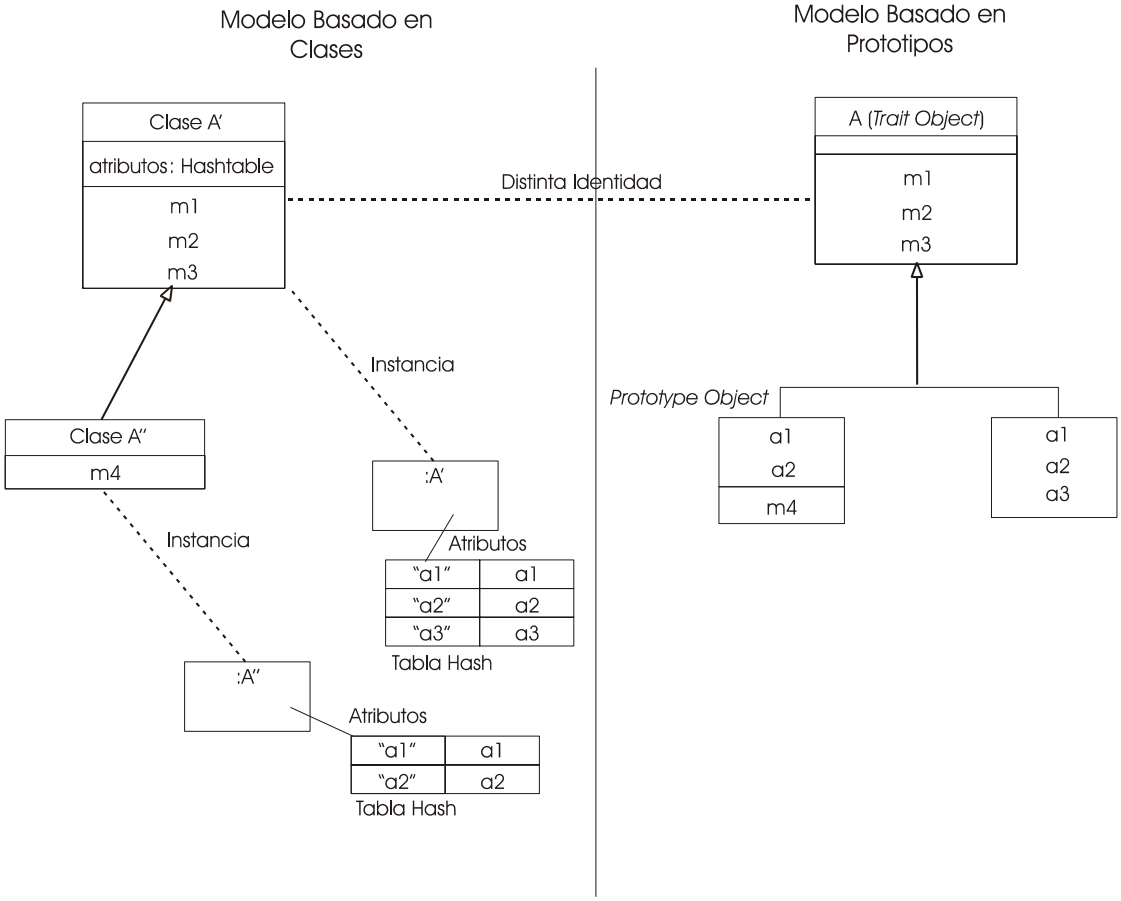

Imagen 11.4: Interoperabilidad clases - prototipos sin mantener la identidad

ESCENARIO 2: TRASLADAR SÓLO ESTADO AL MODELO DE CLASES

Este caso es un caso particular del anterior donde el proceso de copia de métodos no existe y sólo se contempla trabajar con atributos. Esto crearía objetos valor (*value objects*) o *Data Transfer Objects*, usados por *frameworks* de desarrollo de aplicaciones como *J2EE* ([SunJ2EE07]), *Spring* o *Ruby On Rails* [Thomas05] para, por ejemplo, la comunicación entre las distintas capas de una aplicación.

11.2.3 Conclusiones

A modo de conclusión, podemos afirmar que el modelo computacional hibrido definido en nuestro sistema permite la interacción entre ambos tipos de lenguajes. La interoperabilidad que poseía el sistema original haría visibles entidades definidas en un lenguaje desde cualquier otro, pero el nuevo modelo computacional asegura que el uso de entidades definidas en un lenguaje estático desde un lenguaje dinámico, y viceversa,

se haga de manera que ambos tipos de lenguajes puedan operar correctamente sobre ellas y establecer una colaboración sin caer en incoherencias.

12 OPTIMIZACIÓN DINÁMICA

Uno de los principales objetivos a alcanzar en esta tesis es tratar de acercar el rendimiento en ejecución de los lenguajes dinámicos al que ofrecen lenguajes estáticos como *Java* o *C#*. En los capítulos anteriores hemos visto cómo se plantea la modificación, tanto del motor computacional como del modelo, de la máquina para dar un soporte adecuado a los mismos. No obstante, estas modificaciones, que dan soporte nativo a un modelo de orientación a objetos basado en prototipos con inferencia dinámica de tipos, pueden hacer que la máquina baje excesivamente su rendimiento. Por ello, en este capítulo se van a plantear diferentes estrategias para que el rendimiento ofrecido por el sistema extendido aumente, teniendo además en cuenta que la compilación de los programas se hará mediante *JIT.*

12.1 OPTIMIZACIÓN DE LA BÚSQUEDA DE MIEMBROS

Si bien es cierto que un modelo de herencia por concatenación como el poseído por lenguajes como *C++* ofrece menos flexibilidad que un modelo dinámico por delegación, este modelo tiene otro tipo de ventajas. Como en la mayoría de los casos vistos, un incremento de flexibilidad repercute en una disminución significativa en el rendimiento de los programas que la usan, y éste no es una excepción. Como hemos descrito en un capítulo anterior, en un modelo de herencia por concatenación todo objeto tiene una clase asociada y ésta pertenecería a una jerarquía determinada de clases. Cada una de las clases de la jerarquía aportará a la clase final, y por tanto a sus instancias, una serie de miembros mediante herencia.

La principal ventaja de esta disposición de miembros es que las búsquedas que se efectuarán durante la ejecución de un programa cualquiera (operación muy frecuente) podrán optimizarse adecuadamente, ya que toda la información estará en un único punto (la clase final) y en una sola estructura de datos. Por tanto, con esta aproximación no será necesario realizar búsquedas por la jerarquía de clases para localizar miembros, y será posible además ordenarlos de la forma más conveniente en aras a obtener el mejor rendimiento posible.

No obstante, el modelo que se ha planteado en el sistema extendido no puede seguir fielmente esta aproximación. En un modelo de herencia por delegación como el ya descrito, que plantea la relación de herencia como una asociación más y modificable dinámicamente, no sería viable la centralización de toda la información accesible por una clase en ella misma, puesto que cualquier cambio en una clase de su jerarquía implicaría reevaluar todos los miembros de la misma para ver si tiene sentido que sigan formando parte de ella, operación que tendría un coste no desdeñable. Por otra parte, esta forma de guardar los miembros tampoco respondería bien ante las operaciones de modificación estructural de clases e instancias como las que se plantea realizar en esta tesis.

Por ello, al necesitar implementar una búsqueda jerárquica de miembros, se deben investigar alternativas que permitan acelerar en la medida de lo posible este tipo de búsquedas. Por otra parte, de cara al uso operativo de las capacidades reflectivas,

parece razonable afirmar que, en cualquier programa estándar, las operaciones de acceso a miembros serán mucho más numerosas que las de adición / eliminación (que probablemente requieran usar también estas operaciones de acceso para comprobar la validez de las modificaciones que hacen), por lo que serán estas operaciones las que recibirán un mayor esfuerzo de optimización, incluso en detrimento de las otras. Para ello, se han estudiado y evaluado las alternativas que se presentarán a continuación.

12.1.1 Nueva Estructura de la Información en las Clases

Dado que el sistema escogido usa un modelo computacional basado en clases y estático, la forma de estructurar la información relativa a clases e instancias estará determinada por esta circunstancia. Por ello podría resultar atractivo para nuestros propósitos estudiar la implementación de otras plataformas dinámicas en funcionamiento actualmente y tratar de tomar ideas de su arquitectura y optimizaciones para trasladarlas al sistema extendido.

No obstante, esta idea ha de tomarse con precaución. Teniendo en cuenta que el sistema está altamente optimizado de cara al modelo basado en clases, y que no podemos perjudicar o limitar el soporte para este tipo de lenguajes que el sistema ya posee, este tipo de cambios puede desestabilizar o limitar este soporte, por lo que esta vía de actuación deberá ser estudiada muy detenidamente. Por otra parte, la complejidad de hacer cambios de este estilo será probablemente elevada (la implementación de una máquina abstracta puede ser compleja y por lo tanto su modificación), por lo que parece prudente limitar los cambios que se hagan en este sentido, y por tanto esta vía de actuación podría no ser adecuada para lograr optimizaciones significativas a priori.

Otra consideración que debemos hacer al respecto es el grado de integración de la información. Según la estructura de la implementación que sea seleccionada finalmente, y las posibilidades que su diseño ofrezca de cara a su ampliación, será posible o no integrar la nueva información añadida con la ya existente (o estática) y lograr una manipulación homogénea de todos los miembros de clases e instancias, que estarán representados en memoria bajo una misma estructura de datos a la que se accederá siempre de la misma forma, independientemente del origen de los distintos miembros. En caso de lograr esto, la búsqueda de miembros dentro de una clase será sencilla de implementar y se apoyará en los mecanismos ya existentes, al coexistir toda la información dentro de la misma estructura de datos. Si esto no fuera posible, entonces la información añadida por código (estática) y la dinámica permanecerían separadas y podría identificarse claramente el origen de un miembro concreto dado el caso, situación en la que tiene sentido la aplicación de algunas de las posibilidades de optimización que vamos a mencionar a continuación.

12.1.2 Búsquedas Previas de Coste Reducido

Como se ha dicho anteriormente, la operación que se ha juzgado como más costosa en el nuevo modelo creado y que se va a tratar de dar prioridad para su optimización es la búsqueda de miembros por la jerarquía de clases. Ésta constaría de dos partes principales:

- **Recorrido de la jerarquía**: Navegación por la jerarquía de clases de un elemento dado hasta llegar a la clase "padre" `Object`.
- **Búsqueda de miembros**: Dentro de cada clase de la jerarquía, buscar en la información del miembro buscado.

Es posible optimizar la segunda parte de este algoritmo "marcando" las diferentes clases de la jerarquía con un *flag* que indique si en ellas se ha realizado alguna operación reflectiva (añadir, modificar o eliminar miembros), de manera que el algoritmo de búsqueda haga la segunda parte de la operación en estas clases más eficientemente (empleando por ejemplo los mecanismos de búsqueda del sistema original) y de esta forma se pueda incrementar el rendimiento del mismo. Para la implementación de este mecanismo se requerirá pues añadir alguna información adicional a objetos y clases y modificar las operaciones de adición, modificación y eliminación de miembros para contemplar este caso. Nótese que esta técnica de optimización trata de adaptar el algoritmo de búsqueda que se va a usar a la información manejada, y que para que ésta sea viable se debe estructurar la información de la clase para que pueda diferenciarse de alguna manera la información añadida dinámicamente de la que no. Si esto ocurre, es posible la implementación de una primitiva de búsqueda previa a una búsqueda "real", de coste mucho más reducido, que compruebe si alguna clase, perteneciente a la jerarquía de aquella a la que se intenta acceder, tiene alguna información reflectiva asociada o no, caso este último en el cuál se podría hacer uso de las funciones ya existentes en el sistema para acceder a los miembros y lograr alguna mejora potencial de rendimiento, evitándose de esta forma hacer un recorrido completo por toda la jerarquía de clases.

Por último, ha de tenerse en cuenta que en el caso de que se lograse una integración total entre ambas clases de la información (estática y dinámica) de las clases, esta optimización no podría implementarse y se tendría que hacer una búsqueda en cada clase de la jerarquía usando los algoritmos de búsqueda que la máquina original ya tuviese implementados.

12.1.3 Reutilización de Mecanismos Estáticos

Como se ha visto en el capítulo dedicado al diseño del motor computacional, un *JIT* como el que usa el sistema original debe modificarse para hacer una llamada a un código que implemente los principios de funcionamiento de las primitivas reflectivas cada vez que se acceda a un miembro, y que permita al código *CIL* adquirir la semántica del modelo computacional diseñado al efecto. Esta llamada a un código concreto se hará en lugar de la sustitución del miembro por una dirección de memoria nativa, pero esta dirección sigue calculándose, y puede ser pasada al código que gestiona el acceso al miembro para acceder a él directamente en caso de que no haya información dinámica susceptible de modificar la estructura original de las clases e instancias que se creó al compilar el programa.

Por tanto, en el caso de que no se haya hecho ninguna operación reflectiva que afecte a la estructura del elemento al que se accede, el miembro requerido por la operación seguirá estando correctamente designado por la dirección de memoria que el *JIT* calcula, y podremos usar dicha dirección para acceder a dicho miembro de la forma más eficiente posible.

12.1.4 "Anotación" Previa de Clases

La descripción de esta última posible vía de optimización se basa en que el sistema podrá funcionar de manera diferente en función de si se ha hecho o no uso de las primitivas reflectivas. El sistema extendido, ante programas que no hagan uso de las primitivas de reflexión añadidas al mismo, podrá usar todas primitivas que poseía originalmente para acceder a los miembros (empleando por tanto el modelo de concatenación), ya que la información no habrá sufrido cambios respecto a su compilación. Es en el momento en el que se usen estas primitivas cuando la información presente en el sistema variará y se deberá poner especial cuidado en su tratamiento. Por ello, partiendo de la premisa de que una clase que no haya sido afectada por una modificación dinámica podrá usar los medios que el sistema posee para acceder a sus miembros, y que estos medios gozan de la máxima eficiencia posible, es razonable incorporar al sistema un mecanismo de anotación de clases que permita maximizar el rendimiento de las operaciones de búsqueda asociadas a las mismas a costa de penalizar otras operaciones, mucho menos frecuentes y de un impacto por tanto muy inferior en el rendimiento global del mismo.

Para ello se partirá del hecho de que cuando una clase sufra una modificación de cualquier tipo, cualquier clase descendiente de la misma o instancia relacionada podrán "ver" esta modificación. Por tanto, cabe la posibilidad de que el sistema "anote" las clases que van a ser afectadas por cada modificación realizada en una estructura de datos, de manera que en todo momento tenga guardada la información relativa a que clases han sido modificadas dinámicamente. El objetivo de mantener esta información es precisamente el de acelerar las operaciones de búsqueda de manera que, cuando se necesite hacer alguna de estas operaciones sobre una clase, pueda consultarse en esta información si ha sido o no alterada de forma directa o indirecta, reutilizando los mecanismos de acceso originales en el sistema en caso de que no lo haya sido, tal y como se vio en el punto anterior. Nótese que de esta forma se podrá discriminar qué clases pueden optimizarse (usando los medios de búsquedas originales basados en concatenación) y qué clases no con la mayor granularidad posible, logrando así el mejor rendimiento.

Para la implementación de un mecanismo de este estilo es necesario tener claro que, si bien es posible que una clase pueda obtener una referencia a su clase padre, lo contrario es imposible en muchos sistemas, por lo que habría que construir una tabla adicional que mantuviese las relaciones entre una clase y sus descendientes para poder implementarlo. Esta tabla podría rellenarse en la inicialización del sistema, al arrancar el mismo, y cada vez que se cargue alguna clase nueva, minimizando el impacto en tiempo de ejecución. La tabla contendría pues un identificador de cada clase y una relación con clases hijas, así como un *flag* que indicase si ha sufrido o no modificaciones dinámicas. Por último, cabe decir que una forma más sencilla pero menos eficiente de hacer algo similar es poner un *flag* global a nivel de toda la estructura de clases, que indicase si el sistema puede usar las primitivas ya existentes o no. Esto no obstante logrará una optimización inferior, debido a que una operación reflectiva afectaría al comportamiento sobre todas las clases.

12.2 OPTIMIZACIÓN DE LA LOCALIZACIÓN DE NUEVOS MIEMBROS

Las modificaciones planteadas al sistema original requieren un lugar donde guardar adecuadamente toda la información que se añada a clases e instancias, y esta forma de guardar la información debe proporcionar un rendimiento óptimo. Para ellos hay dos posibles opciones, a las que ya se hicieron mención en la sección anterior:

- **Integrar la nueva información que se añada dinámicamente con la existente**: De esta manera, una clase o instancia alteraría dinámicamente el espacio que ocupa en memoria, al integrar dentro de sus estructuras de datos todos aquellos miembros que se le añadan (o eliminar aquéllos que se borren). Esta sería la solución que en principio debería obtener mejores resultados de rendimiento debido a que todos los algoritmos de búsqueda que el sistema original posee podrían seguir usándose (la estructura de datos que mantuviese los miembros no variaría, sólo sería más grande). No obstante, pueden existir problemas con esta aproximación en función de las optimizaciones que se hayan implementado en el mismo. Por ejemplo, es posible que una vez cargada la clase en memoria, al iniciar la ejecución del programa, el motor haga cálculos y optimizaciones partiendo del tamaño en memoria de la clase que tenía en ese momento inicial, por lo que cualquier modificación (que el sistema no espera, ya que está optimizado para lenguajes estáticos originalmente) podría desestabilizar la ejecución del mismo. Cabe la posibilidad de adaptar convenientemente las partes del sistema que hagan uso del tamaño de las clases para sus cálculos y adaptarlos a la nueva situación, pero una modificación de este tipo puede ser demasiado compleja, debido a que pueden ser muchos los puntos a modificar por este motivo y que alterar este parámetro puede tener efectos laterales sobre otros módulos, afectando a la compatibilidad.
- **Situar la información dinámica en entidades independientes**: Esta posibilidad se usaría en el caso en el que la primera no fuese viable, pero estaría condicionada por varias premisas que citaremos a continuación, ya que requieren que el tratamiento que se le dispense sea equivalente al que recibe el resto de la información del motor.

    - La información debe estar asociada unívocamente con su clase o instancia de forma individual y debe poder accederse eficientemente a ella a partir de la misma.
    - La estructura de datos que guarda la información de los miembros dinámicos debe también tener una alta eficiencia en operaciones de búsqueda.
    - Toda la información que se guarde dinámicamente debe estar integrada con el motor del sistema (utilizando sus estructuras de datos, estructuras, mecanismos, etc.)
    - La información debe ser compatible con el recolector de basura del motor de ejecución. A pesar de no ser una parte integrante de la clase o instancia con la que está asociada, la información dinámica debe poder ser tratada adecuadamente por el mismo, al igual que la que el motor ya posee, para que no sea eliminada en un momento no previsto y haga que el sistema falle en tiempo de ejecución.

13 IMPLEMENTACIÓN DEL SISTEMA

Una vez expuestos todos los aspectos relevantes del diseño del sistema, describiremos a continuación una serie de características del estándar sobre el que se va a implementar el soporte para primitivas de reflexión, seleccionando además la implementación del mismo más acorde con los requisitos de esta tesis.

13.1 EL ENTORNO DE EJECUCIÓN CLI

El *CLI* es un estándar *ECMA* (*ECMA-335*) que especifica una arquitectura donde múltiples lenguajes pueden ejecutarse e interoperar sobre una misma máquina virtual [ECMA02]. Haremos una descripción de las principales características del entorno de ejecución *CLI* siguiendo lo expuesto en [Stutz03], para conocerlo más en profundidad y que posibilidades ofrece. En el apéndice A de esta tesis se desarrollarán más en profundidad los conceptos que vamos a enunciar aquí sobre la base de una implementación concreta del estándar (*SSCLI*).

13.1.1 Introducción: Principios del CLI

El estándar *CLI* es una arquitectura orientada a datos, de manera que los datos se construyen de una forma segura en cuanto a su tipo y de forma independiente del lenguaje que los use. Para permitir esto, el sistema usa metadatos que describen el comportamiento y la representación en memoria del *software* desarrollado. Estos metadatos son usados por el *CLI* para permitir que componentes construidos en diferentes orígenes (los distintos lenguajes soportados por la plataforma) puedan colaborar libremente y ser cargados en memoria de forma controlada, pudiendo acceder cada uno de ellos a todas aquellas partes públicas de los otros que requieran, independientemente de los lenguajes en los que hayan sido construidos.

Además del estándar *CLI,* otro estándar importante es el que describe el lenguaje *C#* [ECMA33405], diseñado para sacar el máximo partido de las características del *CLI*. *C#* es un lenguaje orientado a objetos, y aunque muchas veces aparece íntimamente relacionado con el *CLI*, deben considerarse elementos diferentes e independientes.

La ejecución de *software* en el *CLI* la lleva a cabo su motor de ejecución, que contiene la representación de todos los componentes usados por las aplicaciones junto con un conjunto de código adicional que no está basado en componentes pero que es necesario para las diferentes funciones del mismo. Este motor es el encargado de interpretar dinámicamente los metadatos que describen estos componentes. El tipo de ejecución que lleva a cabo el motor hace que el código que maneja se denomine *managed code,* debido a que el entorno de ejecución tiene un amplio control sobre el mismo y puede obtener información del programa en ejecución, así como efectuar sobre él cualquier tipo de operación de chequeo o similar.

Por otra parte, los metadatos son "empaquetados" en los llamados ensamblados (*assemblies*) junto con código de programa en *CIL* (el lenguaje intermedio usado por esta plataforma). A partir de estos ensamblados un programa se puede cargar en memoria y convertirse en código ejecutable apropiado para la arquitectura concreta y el sistema operativo sobre el que se esté ejecutando, mediante una serie de eventos concretos como los que se muestran en la figura 13.1.

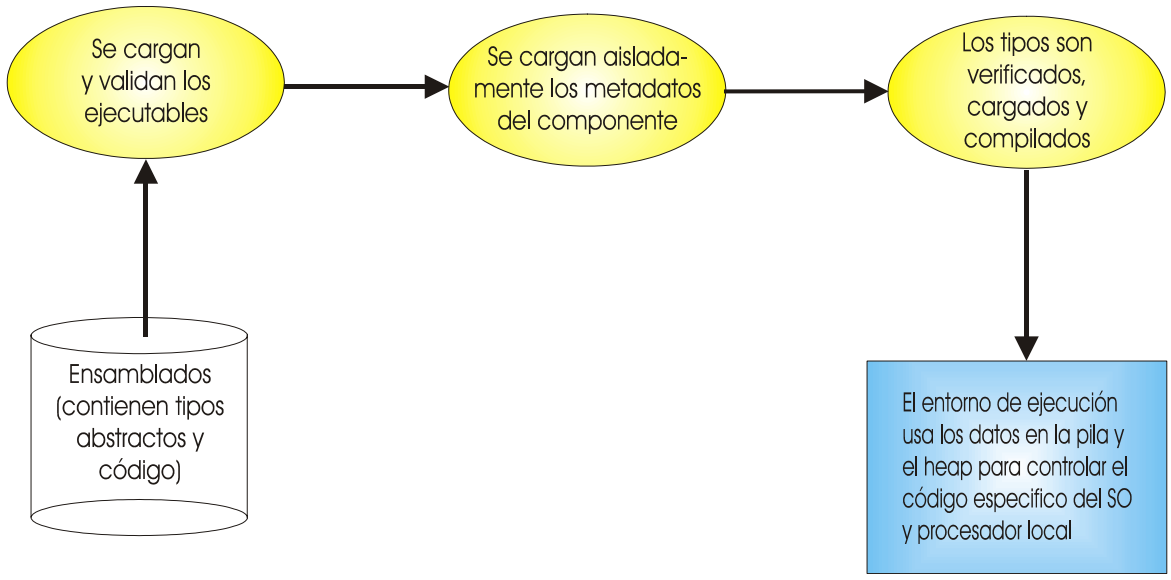

Figura 13.1. Pasos para cargar metadatos de un `assembly` y conversión de los mismos a código ejecutable

El *CLI* agrupa los conceptos tradicionales de compilador, *linker* y cargador, ya que es él mismo el que carga en memoria los datos, compila el código y resuelve los símbolos a los que el código se refiere. Además, la especificación del *CLI* describe de forma pormenorizada y con todo detalle como debería funcionar el *software* gestionado (*managed*) y también cómo puede coexistir e interoperar con código no gestionado, de forma que ambos tipos de código puedan compartir recursos y funcionalidades.

13.1.2 Conceptos Fundamentales de la Especificación CLI

En esta sección se van a exponer una serie de ideas que determinan el diseño, la especificación y el modelo de ejecución del *CLI*. Estos conceptos se plasman en la implementación final en abstracciones y en técnicas concretas para organizar el código:

- Todas las entidades programáticas son expuestas usando un sistema de tipos unificado (*CTS*).
- Los tipos son empaquetados en unidades portables que tengan capacidad autodescriptiva (los ensamblados mencionados anteriormente).
- Los tipos son cargados en memoria de forma que puedan colaborar y compartir sus recursos, pero manteniéndose aislados unos de otros, de manera que no se creen dependencias innecesarias entre ellos.

- La dependencia entre tipos en ejecución se resuelve por un mecanismo flexible, que permite su resolución teniendo en cuenta características adicionales, como pueden ser la versión concreta de un tipo (coexistiendo tipos de diferentes versiones) o diferencias entre culturas (localización cultural del tipo), entre otras.
- El comportamiento de los tipos se representa de manera que pueda comprobarse estáticamente si su uso es seguro y correcto, de forma que se puedan eliminar la mayoría de errores en tiempo de ejecución provocados por el uso incorrecto de un tipo concreto. No obstante, no se impone necesariamente esta restricción a todos los programas que se ejecuten sobre la plataforma.
- Las tareas relativas a la arquitectura *hardware* sobre la que se esté ejecutando el *CLI*, como la compilación a código nativo y la carga en memoria de la información, se posponen hasta el último momento posible, aunque sin impedir la creación de herramientas que decidan hacer este tipo de procesos antes.
- Los servicios existentes en tiempo de ejecución se diseñaron para que estén representados por un formato de metadatos extensible, que permita incorporar nuevos conceptos y cambios en el futuro.

Una vez expuestos los principios del *CLI*, describiremos algo más en detalle algunos de los más importantes.

13.1.2.1 TIPOS

Dado que el *CLI* usa un modelo de orientación a objetos basado en clases tradicional, los tipos son la unidad de organización del código, tanto en estructura como en comportamiento. La aproximación usada en su diseño es muy similar a la empleada en otros sistemas basados en clases similares: Un tipo describe atributos, que se usan para guardar datos (estado), y también métodos, que especifican comportamiento. Tanto estado como comportamiento puede existir a nivel de instancia, donde los componentes comparten la estructura pero no la identidad, como a nivel de tipo, donde todas las instancias comparten una sola copia de los datos o de la información relativa a la ejecución de un método (los tradicionales miembros estáticos). El modelo de componentes soporta además conceptos tradicionales de la orientación a objetos como por ejemplo:

- **Constructores**: Empleados de la misma forma que en otros sistemas similares, para inicializar los valores de una instancia cuando ésta es creada.
- **Herencia**: No hay herencia múltiple, una clase no puede tener dos clases "padre".
- **Polimorfismo**: Basado en el *interface* de los componentes.

Toda la estructura de un tipo se representa en el sistema en forma de metadatos. Estos metadatos siempre están disponibles y accesibles, tanto para el entorno de ejecución como para otros programadores y tipos. Mediante el uso de metadatos, cualquiera de los elementos mencionados puede acceder a la estructura de un tipo concreto y operar en función de lo que contenga la misma, sin necesidad de que exista necesariamente una dependencia entre el tipo y aquella entidad que accede al mismo, con lo que se consigue el propósito anteriormente mencionado de mantener la colaboración entre tipos preservando la independencia entre ellos.

El *CLI* sólo carga un tipo en memoria cuando es necesario, es decir, cada tipo podrá ser compilado y sus dependencias resueltas a demanda del *software* en ejecución

en un momento dado, usándose sólo aquellos tipos requeridos por la ejecución de un programa en un instante dado. Además, toda referencia de un tipo a cualquier otro es simbólica, de manera que inicialmente se guarda un nombre que luego podrá ser resuelto en tiempo de ejecución. Esta forma de referenciar otro tipo permite más flexibilidad que la que soportarían los sistemas que guardasen, por ejemplo, direcciones de memoria, ya que datos que se refieran a zonas físicas de memoria, o a elementos o lugares relacionados con el entorno físico de la máquina sobre la que se ejecuta el *software* limitarían las funcionalidades que se podrían implementar. Un ejemplo son los mecanismos de versionado avanzados, que pueden cargar nuevas versiones de un componente sin necesidad de reconstruir el *software* que lo usa, gracias a una lógica de resolución y enlazado de tipos del entorno de ejecución flexible como la poseída por el estándar *CLI,* que no vincula sus referencias con direcciones físicas de memoria.

Por último, cabe destacar que cualquier tipo puede heredar tanto estructura como comportamiento de otro tipo mediante el mencionado mecanismo de herencia simple. Todos los métodos y atributos no privados del tipo base son incluidos en la definición del tipo hijo, pudiendo por tanto usar instancias del tipo derivado allí donde se usen instancias del tipo base. Este mecanismo es idéntico al ya implementado en otros lenguajes como *Java*, y al igual que en este lenguaje, cualquier tipo puede implementar un número cualquiera de interfaces y todos los tipos derivan, directa o indirectamente, de una clase padre *Object* (`System.Object`.en el *CLI*).

13.1.2.2 INTEROPERABILIDAD:

Una de las características más interesantes del *CLI* es su interoperabilidad, es decir, la capacidad de que código escrito en un lenguaje X de la plataforma pueda ser usado sin restricciones desde otro lenguaje Y sin necesidad de recurrir a complejos mecanismos de adaptación. Para permitir esto, todo lenguaje de alto nivel definido en la plataforma es compilado a un lenguaje intermedio (*Common Intermediate Language o CIL*) que describe el programa, y cada dato formará parte de un sistema de tipos común, que proporciona un conjunto básico de tipos para dicho lenguaje intermedio (*CTS* o *Common Type System*). Ambas entidades forman un modelo de computación abstracto, que a su vez posee una serie de reglas para traducirlo a un conjunto de instrucciones nativas y de referencias de memoria propios de cada sistema.

El empleo de un *CIL* común a todos los lenguajes permite la interoperabilidad. Este lenguaje intermedio posee un conjunto relativamente amplio y rico de instrucciones (*opcodes*) que tratan de no ceñirse a ninguna arquitectura *hardware* concreta, sino que emplea una máquina abstracta de pila como base. El sistema común de tipos (*CTS)*, define un conjunto básico de tipos que permiten ser usados en cualquier lenguaje de alto nivel del sistema y que garantice la interoperabilidad mencionada. Para que esto sea posible, todos los lenguajes de alto nivel tienen unas reglas consensuadas en cuanto al conjunto de instrucciones y de tipos *CIL* que va a usar cada instrucción, por lo que en este sistema un tipo concreto X no tiene un tamaño diferente en función del lenguaje escogido (como ocurre, por ejemplo, en lenguajes como *C*, donde el tamaño ocupado por un tipo concreto puede ser función de la arquitectura o *SO* sobre la que se ejecuta el compilador). Por tanto, a la hora de diseñar un compilador para un nuevo lenguaje de esta plataforma, se deberá definir exactamente a qué instrucciones del *CIL* equivale cada sentencia y también a qué tipo *CTS* equivale cada uno de sus tipos, de forma que todos los tipos definidos en el mismo puedan ser reutilizados en otro lenguaje. Mediante este mecanismo, un tipo T1 definido en un lenguaje L1 podrá ser usado desde otro lenguaje L2, al existir una equivalencia de T1 en el *CTS* que permita a L2 obtener la definición del tipo y emplearlo adecuadamente desde el mismo.

13.1.2.3 ENSAMBLADOS

Como ya hemos visto, dos de los principales conceptos que forman parte del *CLI* son su sistema de tipos y su entorno de computación abstracto, que permiten que diferentes componentes *software*, escritos por diferentes programadores y en diferentes momentos y lenguajes puedan ser verificados, cargados y usados de forma conjunta para formar aplicaciones completas. Para facilitar esta labor, el *CLI* empaqueta los componentes individuales en los mencionados ensamblados (*assemblies*), que pueden ser cargados de forma dinámica dentro del entorno de ejecución a demanda y que pueden tener procedencias diversas: la red, el disco local del sistema o incluso creados dinámicamente por el programa. Los ensamblados son pues los encargados de facilitar este tipo de operaciones, ya que definen la semántica del modelo de componentes del *CLI*, es decir, no pueden existir tipos "aislados" fuera de un ensamblado (aunque esta restricción es menos estricta en la versión 2.0 del *CLI*), ya que éstos son el único medio que el *CLI* tiene para cargar tipos. Cada ensamblado puede consistir en uno o varios módulos (que permiten organizar la información dentro del ensamblado) y un conjunto de metadatos que describe el ensamblado (*assembly manifest*).

Por motivos de seguridad, para evitar que un ensamblado sea modificado o alterado una vez compilado, cada uno de ellos puede ser dotado con una clave criptográfica y una clave *hash* que representan al ensamblado completo y que se guardan en este *manifest* mencionado. El entorno de ejecución comprobará esta información e impedirá la ejecución del código residente en ese ensamblado si la clave *hash* no es correcta, de forma que se evite procesar código malicioso (modificado con el propósito de dañar el sistema por un virus o similar) o bien dañado (por ejemplo, por existir errores de transmisión por red).

Se puede establecer un paralelismo entre los ensamblados del *CLI* y las librerías (*DLL*) de un sistema operativo, ya que ambos son formas de identificar código que por alguna razón es necesario agrupar bajo una misma clasificación. No obstante, el sistema propuesto en el *CLI,* además de ser independiente de cualquier arquitectura, permite que cada componente pueda ser cargado, versionado y ejecutado de forma independiente a otros componentes, aunque dependa de otros, asegurándose además que no se cargarán versiones alteradas o dañadas de dichos componentes. Esto permite además que el sistema siga funcionando aunque los componentes del mismo evolucionen.

13.1.2.4 COMPILACIÓN *JIT*

La especificación del *CLI* estipula la necesidad de separar completamente el proceso de transformación de la representación de los tipos en un lenguaje de alto nivel en una representación intermedia (tipos del *CIL*) de la transformación de dicha representación intermedia en código y estructuras de memoria nativas, específicas del procesador físico donde se va a ejecutar finalmente el código. Este procedimiento acarrea ciertos problemas, como hacer que esta representación nativa produzca código eficiente. Para ello, las implementaciones del estándar incorporan técnicas que tratan de disminuir, tanto en tiempo de ejecución como en el uso de memoria, el coste de ambas transformaciones de manera que se pueda obtener un rendimiento lo más parecido posible a la compilación directa a código nativo. Por ejemplo, los tipos no son cargados en memoria hasta que son necesarios, por lo que, si una aplicación no hace uso de un tipo T, éste nunca ocupará espacio en memoria, mientras que los métodos de los tipos no son transformados a código nativo hasta que no se ejecutan, por lo que si un método no es necesario durante una ejecución concreta, nunca será procesado. En definitiva, se hace uso extensivo de la compilación *JIT,* descrita en un tema anterior, junto con un abanico de técnicas variadas para permitir mejorar el rendimiento final de las aplicaciones que se ejecuten sobre este sistema.

Por tanto, cuando el entorno de ejecución está en marcha, éste se encargará de controlar la transformación de un componente abstracto (en su representación intermedia) en código nativo ejecutable, cargándolo y compilándolo según sea necesario. El proceso típico de compilación, enlazado y carga sigue existiendo, pero bajo un control inteligente del entorno de ejecución, que permite al mismo tiempo disminuir el coste de estas técnicas y tomar el control completo del comportamiento de los componentes que se están ejecutando en un momento dado.

Por tanto, durante todo el proceso de ejecución del *CLI* los componentes representados por el lenguaje intermedio e independiente de la plataforma (*CIL*) serán compilados, cargados y añadidos al sistema, recibiendo éste la nueva funcionalidad aportada por los componentes cargados cada vez que se hace el proceso. El control total del entorno de ejecución sobre estos procesos permite verificar los tipos de los nuevos componentes cargados o abortar la ejecución del componente si no se considera seguro por los mecanismos vistos antes. Todo el mecanismo de posponer la carga y compilación hasta que son necesarias y el control sobre estos procesos forma parte de la ejecución *managed* a la que nos referíamos anteriormente.

13.1.2.5 MANAGED EXECUTION

Cuando se carga un tipo, se lanza a su vez el proceso de control de la ejecución mencionado anteriormente, que ahora se describirá más en detalle. En el proceso de carga, el *CLI* compila, ensambla y valida el formato del ejecutable y los metadatos del programa, validando también los tipos definidos e incluso los recursos (memoria y ocupación del procesador) de los componentes que controla en tiempo de ejecución. Para ello, el *CLI* incluye un soporte para enlazar nombres con las entidades a las que designan, cargarlas en memoria, compilarlas, aislarlas, sincronizar su código y resolver símbolos, controlando cada uno de estos procesos y determinando cómo el código interactúa con el sistema operativo y la máquina sobre los que se ejecuta el programa.

Además, el hecho de posponer la compilación, enlazado y carga permite mejorar la portabilidad entre diferentes sistemas y también los cambios de versiones. Si se posponen las tareas de alineamiento en memoria de los datos, procesamiento de direcciones físicas, uso de las instrucciones del procesador, uso de los convenios de llamada a funciones y el enlazado de los ejecutables a los servicios concretos que ofrece cada sistema operativo, se logrará que los ensamblados gocen de un grado de compatibilidad muy superior. Este proceso es muy robusto cuando se acompaña de una correcta definición de los metadatos que describen lo contenido en los ensamblados y unas políticas de procesamiento del sistema adecuadas.

En lo referente a la seguridad y al control que el entorno de ejecución puede hacer sobre un código concreto, todo ensamblado tiene asociado una serie de permisos que definen lo que puede hacer. Cuando el código de un ensamblado intenta hacer alguna operación comprometida (escribir en un fichero, acceder a la red), el *CLI* puede comprobar en la pila si el código en el ámbito actual tiene los permisos correctos, abortando la ejecución de dicho código en caso negativo.

13.1.2.6 METADATOS

Los componentes del *CLI* son autodescriptivos, es decir, contienen la definición de cada uno de sus miembros y garantiza que esta información esté disponible en tiempo de ejecución permanentemente. Cada tipo, método, atributo o parámetro debe estar completamente descrito en el ensamblado al que pertenece el componente. Como el *CLI* pospone cualquier proceso de carga hasta que es estrictamente necesario, las

herramientas que deseen manipular los componentes o crear nuevas funcionalidades a partir de estos metadatos tienen un alto grado de flexibilidad.

Para obtener información de cualquier tipo, los programadores pueden usar los servicios de reflexión que el entorno de ejecución ofrece, poseyendo toda la funcionalidad descrita en el capítulo dedicado a introspección.

13.2 ELECCIÓN DE CLI COMO PLATAFORMA BASE

Como ya se ha dicho anteriormente, de entre todos los sistemas presentados, se ha seleccionado la plataforma *CLI* para el trabajo que se va a desarrollar en esta tesis. Esta elección se ha basado en las ventajas que las características concretas de este estándar podrían reportar al desarrollo que se va a llevar a cabo. No obstante, para completar la justificación de dicha elección, examinaremos las características del sistema de acuerdo con los requisitos planteados en esta tesis. Describiremos pues cómo se adaptan dichos requisitos a la plataforma *CLI*, para posteriormente pasar a describir más detalladamente las diferentes implementaciones del mismo disponibles actualmente (*Mono*, *DotGNU Project*, *SSCLI* y *CLR*), escogiendo finalmente una de ellas de manera justificada, teniendo nuevamente en cuenta los requisitos. A continuación haremos pues una descripción de hasta qué punto cualquier sistema basado en el *CLI* debería adaptarse a los requisitos planteados en esta tesis, comentando sólo aquéllos que puedan atribuírsele a la plataforma base y no al desarrollo que se va a llevar a cabo.

13.2.1 Requisitos Arquitectónicos

- Máquina virtual: Todos los sistemas basados en el *CLI* considerados trabajan sobre una máquina virtual estática. A la hora de escoger una implementación del mismo, deberemos pues tener en cuenta qué implementación proporciona características como las siguientes de una forma más completa:

    - Estabilidad general del sistema.
    - Estado de desarrollo (se escogerán versiones finales y que no estén inmersas en un proceso de desarrollo y actualización continuo, para evitar incompatibilidades de las nuevas versiones con nuestro código).
    - Capacidad para ser modificada (no existan impedimentos técnicos o legales para ello).
    - Existencia de documentación
    - Rendimiento (incluyendo optimizaciones en tiempo de ejecución) de las aplicaciones finales.

  Implementaciones como *Mono* o *SSCLI* poseen la mayoría de estas características y son potencialmente seleccionables.

- Independencia del lenguaje de programación: Todos los sistemas basados en el *CLI* soportan múltiples lenguajes, diseñados específicamente para la plataforma. Estos sistemas permiten además ampliar el conjunto de lenguajes que se ejecutan sobre ellos (permitiendo por tanto incorporar en un futuro lenguajes dinámicos sobre el sistema extendido) sin que existan a priori barreras al respecto.
- Independencia del problema: La plataforma *CLI* no está orientada a resolver problemas concretos, sino que es un entorno de desarrollo de propósito general.
- Soporte para capacidades reflectivas: El estándar que especifica la plataforma *CLI* incorpora características reflectivas de forma limitada (introspección y programación generativa), aunque superior a otros sistemas similares, como *JVM*.
- Independencia de la Plataforma y Portabilidad: El estándar *CLI* no está en principio vinculado a ninguna plataforma o sistema operativo concreto. De todas formas, a la hora de seleccionar una implementación, se debe tener en cuenta el conjunto de plataformas para las que existen distribuciones de la misma y el coste de migrarlo a otras plataformas (lo que dependerá de cómo esté realizada su arquitectura).
- Interoperabilidad: Como hemos visto, todos los sistemas basados en el *CLI* soportan esta característica, lo que permitirá que todos los lenguajes diseñados para el mismo puedan usar programas desarrollados con otros lenguajes. Esta característica es muy importante para nuestra tesis (ya que mediante la misma podremos lograr un modelo de cooperación entre lenguajes estáticos y dinámicos) y la plataforma *CLI* es el único sistema de los estudiados que la usa activamente, lo que repercute muy favorablemente sobre la elección del mismo.
- Extensibilidad: Siendo el *CLI* un estándar publicado libremente, en principio no habría problemas en examinar sus características concretas para poder diseñar ampliaciones. No obstante, las posibilidades de actuación al respecto pasarán por el soporte para ampliaciones ofrecido por las implementaciones concretas de este estándar. Deben pues distinguirse claramente aquéllas cuya implementación es cerrada, y no se dispone de su código fuente, de otras que ofrezcan su código fuente de forma libre o con alguna licencia que imponga alguna obligación razonable o que se pueda cumplir fácilmente. Por tanto, aquellas implementaciones que no permitan disponer libremente del código fuente no serán consideradas, ya que no podremos efectuar ninguna labor con las mismas. Además, debe considerarse también qué otras herramientas de soporte ofrecen para facilitar la extensibilidad de cualquier parte del sistema (herramientas de compilación, documentación, ejemplos, legibilidad del código, etc.) y si existe o no alguna limitación en las modificaciones que se pueden llevar a cabo. En general podemos afirmar que dentro de las implementaciones del estándar *CLI* existen algunas que cumplen con todos los requerimientos de extensibilidad necesarios.
- Proyección del sistema para el desarrollo de aplicaciones reales: Siendo el *CLI* un estándar de una plataforma de propósito general, todas sus implementaciones deberían estar capacitadas para desarrollar aplicaciones reales de uso profesional. No obstante, las diferentes funcionalidades proporcionadas por las distintas implementaciones deberán ser examinadas para ver hasta que punto se permite el desarrollo de determinados tipos de aplicaciones. El *CLR* de *Microsoft* es, sin lugar a dudas, la implementación más extendida de este estándar, siendo base para el desarrollo de un gran número de aplicaciones de diferente naturaleza hoy en día.

Por último, cabe decir que, dado que cualquier implementación del *CLI* es un reflejo de un estándar común, teóricamente nada impediría portar las modificaciones realizadas a otra implementación, incorporando los conceptos vistos a la arquitectura concreta de la implementación deseada. Por ello, puede ser conveniente buscar la implementación que permita desarrollar en un principio las modificaciones necesarias más fácilmente, para que luego pueda servir de modelo

y, en caso de no tener la proyección necesaria para este requisito, portar las modificaciones a un sistema con más extensión cuyo grado de similitud arquitectónica con el seleccionado sea elevado.

13.2.2 Requisitos de Rendimiento

La elección del *CLI* frente a otros sistemas en términos de rendimiento no es sencilla. El rendimiento ofrecido por las implementaciones existentes del *CLI* es aceptable, pero es complicado establecer si dicho rendimiento es "mejor" que el ofrecido por otros sistemas similares (como *JVM*) debido principalmente a las diferencias entre las arquitecturas de ambos sistemas y la construcción de los mismos. La que probablemente sea la implementación del estándar *CLI* más eficiente actualmente, el *CLR* de *Microsoft*, se ha podido valer de la experiencia del diseño de la máquina virtual de *Java* para mejorar algunos de sus aspectos en términos de rendimiento, pero en general es complicado establecer un "ganador" general en estos términos, debido a que normalmente las características particulares de cada uno de ellos les hacen mejores en diferentes tipos de pruebas de rendimiento que se puedan realizar [Doederlein03]. La existencia de versiones especializadas de la máquina virtual de *Java* dificulta aún más dicha elección. Por ello, se ha decidido considerar a ambos sistemas similares en cuanto a rendimiento y dejar que la elección de uno u otro pueda recaer sobre otros factores igualmente importantes.

En lo referente a que sistema basado en el *CLI* es más eficiente, existen estudios comparativos de todos ellos en varios campos [Jeswin06], que pueden permitir establecer un criterio al respecto, aunque no son estudios que permitan hacer afirmaciones definitivas en este aspecto. De todas formas, el rendimiento del sistema base no debe ser un criterio capital para la elección del sistema, ya que sus posibilidades de ampliación son tan importantes como su rendimiento base (no nos es de utilidad un sistema muy eficiente si no es posible su modificación).

13.3 IMPLEMENTACIONES DEL CLI

Siendo el *CLI* un estándar que se adecua a las necesidades planteadas en esta tesis, actualmente hay cuatro implementaciones principales del mismo que se pueden considerar a la hora de usarlas como base para el trabajo a desarrollar. De esas cuatro, se descartará directamente el *CLR* de *Microsoft*, ya que no existe soporte alguno para su modificación interna, al ser un sistema comercial cuyo código no está disponible. Se describirán a continuación las tres alternativas restantes, para finalmente hacer un estudio comparativo que justifique la elección final de la implementación a usar teniendo en cuenta las ventajas y desventajas de cada una de las implementaciones.

13.3.1 CLR y SSCLI

En el año 2001 *Microsoft* liberaba el *SSCLI* [Rotor05] (*Shared Source CLI*, conocido también como *Rotor*), un código disponible gratuitamente y que a la vez era

modificable y redistribuible. Este código contiene una implementación plenamente funcional del motor de ejecución *CLI*, junto con un compilador del lenguaje *C#*, una serie de librerías de programación y una colección de herramientas de desarrollo. Esta distribución fue desarrollada conjuntamente con la versión comercial del *CLI* de *Microsoft*, el *CLR* (*Common Language Runtime*) [MSDNCLR06], que actualmente es la base de la línea de productos *Visual Studio* modernos [VisualStudio06] (de hecho puede decirse que el *SSCLI* es una versión "recortada" de esta versión comercial). La política de desarrollar y hacer pública una versión paralela de una herramienta comercial de forma gratuita tiene los siguientes propósitos:

- Permitir comprobar la portabilidad del estándar *CLI*.
- Permitir estudiar y conocer el entorno de *Microsoft* comercial *CLR*, dado el alto grado de similitud entre ambos entornos.
- Favorecer los proyectos de investigación y académicos sobre el *CLI*, es decir, favorecer proyectos como el presentado en esta tesis.
- Establecer una guía para entender el estándar *CLI* y además poder servir como modelo para otras implementaciones del estándar que se quieran desarrollar.

La licencia *Shared Source* [SSCLI06] otorgada por *Microsoft* a este producto garantiza al usuario los derechos suficientes para implementar y probar sus modificaciones sin limitaciones que puedan condicionar el trabajo desarrollado.

Como se muestra en la figura 13.2, *SSCLI* es un superconjunto del estándar *CLI*, implementando dicho estándar y otras funcionalidades adicionales. A su vez el entorno comercial *CLR* es un superconjunto de *SSCLI*.

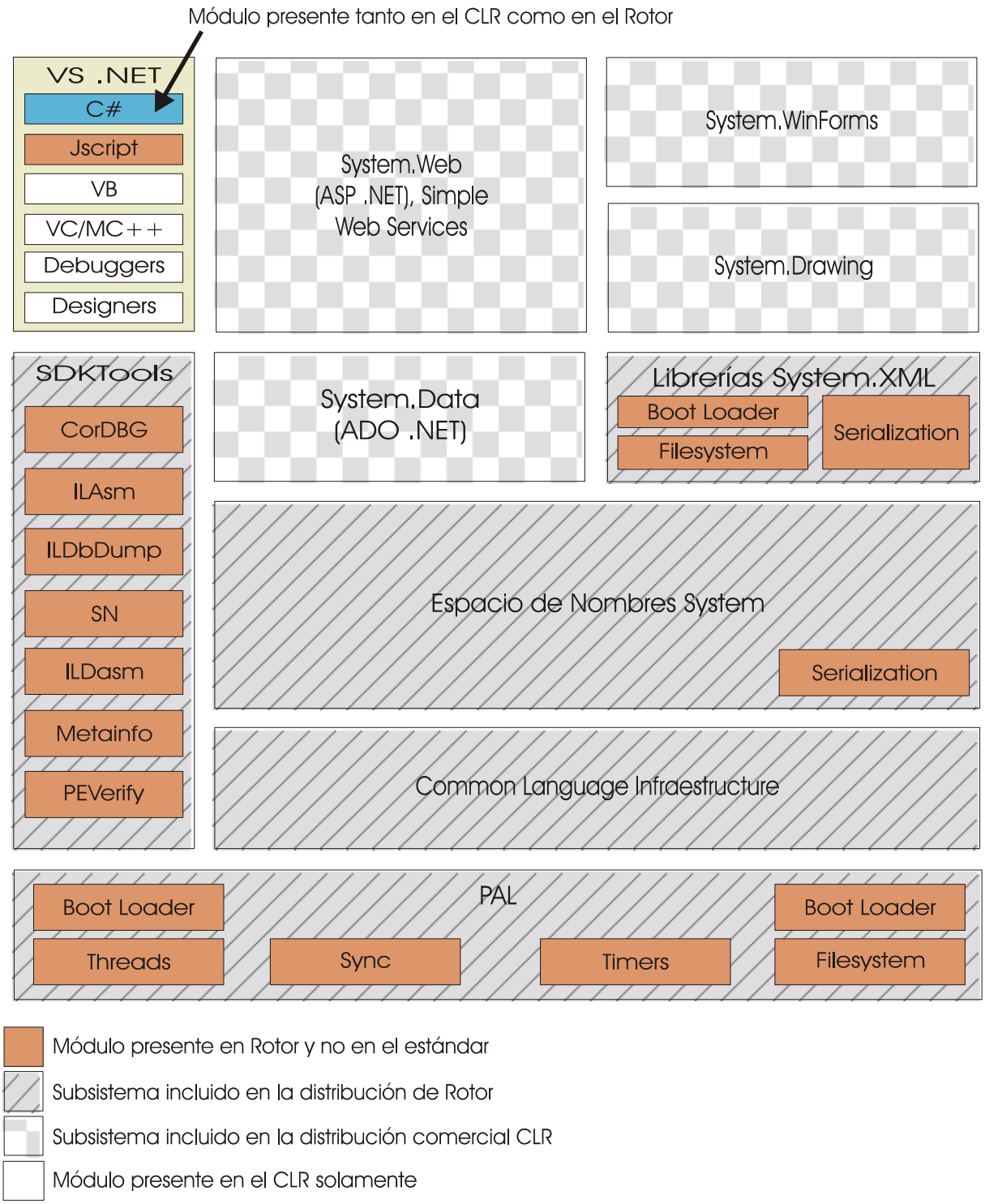

Figura 13.2 Componentes del *SSCLI* y *CLR*

SSCLI en sí es un sistema de tamaño muy grande (aproximadamente unas 3.6 millones de líneas de código), cuyo código posee una gran complejidad, y construido usando una combinación de *C++* (para la construcción del entorno de ejecución, máquina virtual, compilador *JIT*,...) y *C#* (para la construcción de las librerías estándar fundamentalmente). Para construir la distribución hay que seguir un procedimiento en tres pasos: Un compilador nativo de la plataforma de destino de *C++* se encarga de construir la *PAL* (descrita a continuación), posteriormente una serie de herramientas de desarrollo, incluido el compilador de *C#*, son construidas usando la *PAL* que se obtuvo del

paso anterior, y finalmente la combinación de estas herramientas y la *PAL* se encargan de construir el resto de la distribución. La estructura general de *SSCLI* se presenta en la siguiente figura, que iremos describiendo a continuación, teniendo en cuenta que a lo largo de esta sección y en adelante todo lo mencionado acerca de *SSCLI* estará referido a su versión 1.0:

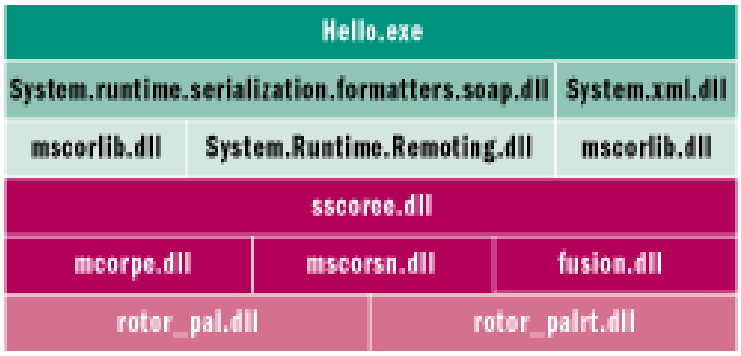

Figura 13.3. Estructura en capas del *SSCLI* [MSDNSSCLI06]

13.3.1.1 La *PAL* (*Platform Adaptation Layer*):

Dado que *SSCLI* está pensado para poder portarse a diferentes plataformas, se le ha provisto de medios que faciliten precisamente esta operación. El propósito principal de la capa *PAL* es aislar a la implementación del resto de las capas de los detalles concretos de los diferentes sistemas operativos sobre los que puede ejecutarse el *SSCLI*. Al ser un producto *Microsoft* y empezar siendo un desarrollo para el sistema *Win32*, la *PAL* presenta un subconjunto de llamadas al *API* de *Win32*. En este subconjunto no se incluye la totalidad del *API* de *Win32*, sólo aquellas partes que son usadas por el *CLI*.

Por tanto, si se desea portar *SSCLI* a una nueva plataforma, se debe empezar por modificar la *PAL* ya que, como se ha mencionado anteriormente, todas las herramientas usadas para construir la distribución de *SSCLI* dependen de esta capa para obtener los recursos necesarios para su funcionamiento. La versión 1.0 de *SSCLI* está preparada para ejecutarse sobre entornos *Windows*, *FreeBSD* y *Mac OS X*, aunque la versión 2.0 sólo es capaz de ejecutarse sobre *Windows XP SP2*.

Uno de los aspectos más interesantes de la *PAL* es que provee al motor de ejecución de capacidades de control. El *SSCLI* está diseñado para funcionar de forma cooperativa con código nativo contenido en procesos del sistema sobre el que se está ejecutando, lo que requiere establecer un control sobre las llamadas que se hacen al sistema operativo para poder implementar correctamente servicios como seguridad o recolección de memoria. El uso de la *PAL* para estas tareas es crítico, ya que el resto del sistema se construye encima de las abstracciones que la *PAL* proporciona, y sin ellas, o sin una implementación correcta de las mismas, el sistema no podrá funcionar correctamente.

13.3.1.2 El Entorno de Ejecución SS*CLI*

El motor de ejecución es el corazón del *SSCLI*. Contiene el modelo de componentes, los servicios ofrecidos en tiempo de ejecución (tratamiento de excepciones, manejo automático de pila y *heap*,...). Elementos como la compilación *JIT*, el sistema de manejo de memoria, el cargador de tipos y ensamblados, la resolución de

tipos, el tratamiento de metadatos y otros mecanismos fundamentales descritos anteriormente se implementan aquí. El entorno de ejecución se construye como un conjunto de librerías que se cargan en tiempo de ejecución y no como un solo ejecutable.

13.3.1.3 LIBRERÍAS EN EL *SSCLI*

Además de todas las capacidades de bajo nivel ya descritas en el estándar (metadatos, *CIL*, *CTS*,...) *SSCLI* también incluye librerías de alto nivel que facilitan la productividad con los lenguajes implementados es esta distribución. Las librerías proporcionadas se pueden dividir en los siguientes grupos:

- **Librerías de productividad**: Formateo de textos, expresiones regulares, colecciones, red, *E/S*, *XML*,...
- **Librerías del entorno de ejecución**: Trazas de pila, recolector de basura, *handles*, hilos, excepciones, seguridad, reflexión, persistencia,...
- **Librerías de tipos**: Tipos primitivos, tipos por valor, cadenas, *arrays*,...
- **Librerías numéricas**: Números decimales, de precisión doble o simple,...
- **Soporte a lenguajes de programación**: Servicios del compilador,...

Estas librerías son un *interface* para acceder los servicios del sistema operativo, pero orientadas a aprovecharse de los servicios del *SSCLI*. Además, facilitan la programación orientada a componentes.

13.3.1.4 MODIFICACIÓN DE *SSCLI*:

Todo el código de *SSCLI* está disponible para su descarga y se incluyen herramientas adecuadas para su manipulación e instalación correcta. La versión disponible para ser modificada puede considerarse estable, ya que no se le han realizado cambios significativos desde el año 2002, lo que permite trabajar sobre el código sin peligro de que futuras modificaciones al sistema base alteren las modificaciones hechas al mismo. Posteriormente a la realización del todo el trabajo planteado en esta tesis (año 2006) ha aparecido la versión 2.0 de *SSCLI* [MicrosoftSSCLI06b].

13.3.2 Proyecto DotGNU

El proyecto *DotGNU* [DotGNU05] es una alternativa a la plataforma *.NET* de *Microsoft*, creado con la intención de evitar el monopolio de esta tecnología por esta empresa. Este proyecto consiste en un conjunto de herramientas *software* que ejecutan aplicaciones diseñadas para el estándar *CLI* sobre diferentes sistemas operativos y plataformas, con herramientas como el servidor de aplicaciones *Web DGEE* y el soporte para el desarrollo de servicios *Web pdpGroupWare*, así como la implementación propia del estándar *CLI,* denominada *Portable .NET*. Este entorno de desarrollo fue inicialmente desarrollado para *GNU/Linux*, pero actualmente existen versiones para múltiples sistemas operativos como *Windows*, *NetBSD*, *FreeBSD*, *Solaris*, y *MacOS X* entre otros, asegurando así su portabilidad. Adicionalmente, soporta varias arquitecturas como *x86*,

PPC, *ARM*, *Sparc*, *s390*, *Alpha*, *ia-64*, y *PARISC*.

Una de las principales motivaciones del proyecto *DotGNU* es la total compatibilidad tanto con los mencionados estándares *ECMA* correspondientes al lenguaje *C#* y al *CLI* ya vistos, como con la implementación comercial del *CLI* de *Microsoft* (el *CLR*), de forma que una aplicación desarrollada para uno de los sistemas funcione correctamente en el otro y viceversa.

13.3.2.1 EL ENTORNO DE EJECUCIÓN DE *DOTGNU*

Este entorno de ejecución es usado para interpretar aplicaciones diseñadas para el estándar *CLI*. Para mejorar la eficiencia ofrecida por la interpretación del *bytecode CLI* generado, este sistema toma la determinación de convertir el *bytecode CLI* en un conjunto de instrucciones más simple, al que llama *CVM* (*Converted Virtual Machine*). Este conjunto de instrucciones simplificadas es entonces ejecutado por un intérprete altamente optimizado. Aunque sin rechazar la construcción de un *JIT* completo para el sistema en un futuro, tal y como hacen las otras implementaciones aquí expuestas, actualmente sólo el método de ejecución mencionado está operativo. Las posibles ventajas de este sistema son varias, siendo la más destacable el poder construir los *opcodes* para adaptarse mejor a las diferentes arquitecturas (32 *bits* frente a 64, por ejemplo) y hacer el código fuente del motor de ejecución muy portable a otras plataformas. En algunas plataformas incluso se llega traducir el código a nativo para una ejecución directa.

En lo referente al compilador, éste es un *software* modular que soporta *C#* (*ECMA-334*) y *C* (*ANSI C*), aunque el soporte para otros lenguajes como *Java* o *Visual Basic .NET* está en desarrollo. Además el compilador está diseñado para soportar múltiples *bytecodes* (actualmente sólo el *bytecode CLI* está disponible, pero en el futuro se espera lograr soporte para otros). El sistema cuenta también con un recolector de basura [Boehm06] para liberar al programador de tareas de gestión de memoria.

Por último, una ventaja de esta implementación frente a las demás es el paquete `System.Windows.Forms`. En lugar de codificar un acceso a librerías ya existentes (como *Gtk* en *Mono*) o eliminar completamente esa funcionalidad (*SSCLI*), esta implementación construye una capa básica de dibujo sobre la que el resto de elementos gráficos se crean y representan. En la práctica, se simula la apariencia visual y el comportamiento de *Windows* en sistemas que inicialmente no siguen esa aproximación. El sistema de ventanas de *DotGNU* es un *software* en proceso de construcción.

13.3.2.2 MODIFICACIÓN DE *DOTGNU*:

El código fuente de esta implementación está completamente disponible para su descarga, así como herramientas para su modificación e instalación. No hay problemas de licencias en cuanto a las modificaciones a realizar, pero es posible encontrar dificultades adicionales (al igual que en *Mono)*, ya que esta implementación es un desarrollo no terminado, en evolución, y puede ocurrir que cualquier cambio que introduzcamos al sistema pueda verse afectado en las siguientes versiones del mismo. Si una nueva versión modifica alguna zona que previamente se ha alterado para incorporar alguna funcionalidad, esto podría obligarnos a duplicar el trabajo realizado o bien a rehacer los cambios para adecuarlos a las características incorporadas en la versión siguiente. Si, para evitar esto, renunciamos a usar versiones más modernas, entonces no podremos aprovecharnos de nuevas características, correcciones de errores, etc.

13.3.3 Proyecto Mono

El proyecto *Mono* [Mono05], como ya se ha dicho, es una iniciativa patrocinada por *Novell*, cuya intención inicial es dar un soporte en código abierto para *UNIX* de la plataforma de desarrollo *.NET* de *Microsoft*. Su propósito es permitir a los desarrolladores de *UNIX* construir y usar aplicaciones *.NET* realmente multiplataforma. Adicionalmente, hay versiones disponibles de *Mono* específicas para *Linux* y la plataforma *Windows*. Este proyecto implementa también los mismos estándares *ECMA* desarrollados en la implementación de *Microsoft*. No obstante, *Mono* hoy en día contempla más tecnologías, de las que a continuación se detallan sólo algunas:

- ***Remoting.CORBA*** [RemotingCORBA05]: Una implementación de la tecnología *CORBA* especialmente diseñada para *Mono*.
- ***Ginzu***: Una implementación del mecanismo de *remoting ICE* [ICE05] (*Internet Communications Engine*). *ICE* es una moderna alternativa a *CORBA* o *COM* para implementar un mecanismo de *remoting*, que además soporta lenguajes como *C++*, *Java*, *Python* o *C#*, entre otros.
- ***Gtk#*** [GTK05]: Soporte para emplear la librería *Gtk*, que es un conjunto de utilidades para la construcción de *GUI* (*Graphical User Interface*) para sistemas *UNIX* y *Windows*. También se soportan las librerías *Diacanvas-Sharp* y *MrProject*.
- ***#ZipLib*** [ZipLib05]: Una librería que permite manipular varios tipos de ficheros comprimidos como *zip* o *tar*.
- ***Mono.Data***: Soporte para trabajar con numerosas bases de datos, como *PostgreSQL*, *MySql*, *Oracle*, *Tds* (protocolo *SQL server*), entre otras.
- ***Mono.Cairo***: [Cairo05] Soporte para emplear el motor de *render Cairo*, que además sirve para implementar las funcionalidades gráficas de dibujo de *Mono*.
- ***Mono.Http***: Soporte para crear servidores personalizados *HTTP* y manejadores *HTTP* comunes para las aplicaciones.

Como puede apreciarse, la implementación de *Mono* no se ha limitado a contemplar sólo las tecnologías del estándar *CLI*, sino que trata de expandir su uso y funcionalidad empleando un amplio abanico de tecnologías adicionales. Además de poseer estas tecnologías, la principal diferencia existente entre esta implementación y la de *Microsoft* es la carencia de tecnologías propietarias de *Microsoft*, como *Passport*. En lo referente a la construcción de *software*, *Mono* ofrece los siguientes componentes:

- Una máquina virtual que cumple con el estándar *CLI* y que contiene (entre otros), un cargador de clases, un compilador *JIT* y recolección de basura.
- Una librería de clases que asegura la interoperabilidad funcionando con cualquier lenguaje diseñado para el *CLR*. Se incluyen tanto las librerías de clases de *CLR* como las de *Mono*.
- Un compilador para el lenguaje *C#*.

Adicionalmente, en *Windows* existen múltiples compiladores de lenguajes que están diseñados para trabajar con la máquina virtual, como *Managed C++*, *Java Script*, *Eiffel*, *Perl*, *Python*, *Scheme* y *Smalltalk*, por citar algunos de los más representativos.

13.3.3.1 EL ENTORNO DE EJECUCIÓN DE *MONO*:

La máquina virtual que *Mono* implementa sigue exactamente el estándar *CLI*. Además, implementa dos tipos de compiladores, un compilador *JIT* (*Just In Time*) y un compilador *AOT* (*Ahead of Time*). También posee un cargador de librerías, un recolector de basura, un sistema de manejo de hilos y la funcionalidad necesaria para asegurar la interoperabilidad. Adicionalmente, *Mono* soporta dos motores de ejecución:

- ***mono***: Un generador de código *JIT* y *AOT*, que asegura el máximo rendimiento.
- ***mint***: El intérprete de *Mono*, un motor de ejecución más fácilmente portable.

Además el motor de ejecución de *Mono* puede ser usado como un solo proceso o bien puede ser incluido en las aplicaciones, lo que permitiría por ejemplo ampliarlas empleando *C#* mientras se mantiene el código existente en *C* o *C++*.

En cuanto a la compilación, se han hecho avances en la mejora de rendimiento del compilador *JIT* y también existe la opción del compilador *AOT*, que permitiría a los desarrolladores precompilar su código a código nativo, mejorando el rendimiento en el arranque y en ejecución (ya que se empleará el proceso de *JIT* menos frecuentemente). De esta forma, será posible precompilar el código de un ejecutable y hacer que el compilador *JIT* ajuste el código para que sea más eficiente en la *CPU* particular donde se esté ejecutando. También se permiten controlar las optimizaciones que se llevan a cabo durante la generación de código, de forma que sea posible activarlas, desactivarlas o trasladarlas desde la compilación *JIT* a la *AOT*. El motor de *Mono* soporta arquitecturas de 32 y 64 *bits* como *x86*, *PowerPC*, *x86-64*, *SPARC*, *SPARCv9* y *S390*.

Al igual que *SSCLI*, *Mono* soporta un *GAC* (*Global Assembly Cache*), usado para compartir librerías entre diferentes aplicaciones o para mantener diferentes versiones de la misma librería instaladas sin incurrir en conflictos de nombres. Por último, cabe decir que actualmente *Mono* usa un recolector de basura conservativo de Boehm [Boehm06], que es menos eficiente que el existente en la plataforma *.NET* de *Microsoft*. En un futuro se espera adoptar otro *GC* de rendimiento comparable al de *.NET*.

13.3.3.2 MODIFICACIÓN DE *MONO*:

Mono está mantenido y desarrollado por una comunidad de desarrolladores, y hasta el momento es un producto de gran tamaño en continua evolución [MonoRoad05], estando algunas de sus partes en un estado inestable o incompleto actualmente. Posee una funcionalidad lo suficientemente amplia y probada como para poder ser una plataforma de desarrollo válida para múltiples tipos de aplicaciones. En cuanto a las capacidades de modificación, todo el código fuente está disponible para el programador que desee examinarlo y/o modificarlo, junto con instrucciones para su correcta instalación y manipulación. No obstante, tal y como se decía en el sistema anterior, al no estar finalizado se debe tener en cuenta que cualquier modificación que se haga al código de *Mono* puede interferir con versiones posteriores del mismo, ya que una nueva versión puede haber modificado un módulo del sistema del que dependía o sobre el que estaba construida alguna de nuestras modificaciones, afectando pues al trabajo realizado. *Mono* tampoco está actualmente en una versión lo suficientemente completa o estable como para no adoptar nuevas versiones a medida que vayan apareciendo, ya que entonces no dispondríamos de nuevas funcionalidades, optimizaciones o correcciones.

13.3.4 Selección de una Implementación como Base de Trabajo

A la hora de seleccionar una implementación del *CLI* como base para el desarrollo, se ha de atender a las características concretas del sistema que puedan afectar al desarrollo de la modificación planteada en esta tesis, sin basarlo únicamente en el número, calidad o potencia de las funcionalidades adicionales añadidas a cada sistema, que por otra parte podrían dificultar los cambios a realizar (ya que potencialmente habrá más subsistemas que tengamos que controlar a la hora de hacer las modificaciones pertinentes, de forma que no se rompa la compatibilidad o se provoquen efectos laterales que los desestabilicen). De este modo, aunque tanto *Mono* como *DotGNU* ofrecen paquetes de *software* que dotan al sistema de más potencia y versatilidad que *SSCLI*, dichas funcionalidades adicionales no están relacionadas con las modificaciones que se pretenden hacer, ni podrían mejorar su implementación o ser aprovechadas para su desarrollo, por lo que no les daremos una importancia capital a la hora de hacer la elección de la implementación final.

En aspectos relacionados con el rendimiento y eficiencia, al ser tanto *Mono* como *DotGNU* sistemas en constante desarrollo, estos parámetros resultan muy difíciles de medir de forma precisa actualmente, ya que las constantes evoluciones y refinamientos de sus arquitecturas hacen que de una versión a otra su eficiencia pueda cambiar significativamente. Existen estudios empíricos que demuestran que *Mono* es más eficiente que *SSCLI* en el proceso de operaciones aritméticas [Vogels05], básicamente debido a que el *JIT* implementado en *SSCLI* no hace énfasis en la optimización del código que genera debido a que su construcción está más bien orientada a que pueda modificarse más fácilmente. No obstante, la simplicidad del recolector de basura de *Mono* frente al implementado en *SSCLI* hace que *SSCLI* sea potencialmente más eficiente en cuanto al uso de la memoria del sistema. Por todo ello, es muy complicado establecer que implementación es "mejor" en cuanto a rendimiento base.

Respecto a la estabilidad del sistema, es decir, el conjunto de potenciales modificaciones que se harán al mismo (evoluciones, actualizaciones, etc.) durante el periodo en el que esté modificándose, la estabilidad mostrada por la distribución *SSCLI* la hace más adecuada para esta tarea que las demás, ya que, como se mencionó anteriormente, los cambios realizados para cumplir los requisitos de esta tesis pueden verse afectados por alguna actualización del sistema base, y por tanto se corre el riesgo de hacerlos inservibles si se quiere usar una versión más moderna del sistema.

Por último, ninguna de las licencias que presentan estos sistemas a la hora de trabajar con su código fuente (*Shared Source* en el caso de *SSCLI*, una licencia *MIT X11* para *Mono* y *GPL* para *DotGNU*) afecta al desarrollo de las modificaciones de esta tesis.

Por tanto, aunque en principio los tres sistemas son igualmente válidos para el desarrollo que se va a plantear en esta tesis, una vez expuestas las razones anteriores el sistema base seleccionado será *SSCLI*, de *Microsoft*, por tres motivos principales:

- Es un sistema no sometido a constantes evoluciones y cambios en su código, que permitiría concentrarse sólo en los cambios realizados sin necesidad de atender a nuevas evoluciones constantemente. La versión utilizable lleva en funcionamiento sin prácticamente cambios desde el año 2002.
- Se ajusta perfectamente a las necesidades pedidas en cuanto a facilidades de modificación, y no cuenta con un gran número de funcionalidades adicionales que puedan plantear problemas al hacerle ciertas modificaciones (es un sistema más simplificado y orientado a hacer modificaciones).
- El alto grado de similitud con el entorno comercial *CLR* hace que las modificaciones

realizadas sobre este sistema puedan ser trasladadas a este entorno comercial de una forma teóricamente más sencilla.

- La modificación planteada en esta tesis ha sido premiada en el "*Second Microsoft Research SSCLI (Rotor) Request for Proposals (RFP)*" como el proyecto "*Extending Rotor with Structural Reflection to Support Reflective Languages",* dotando al mismo de soporte económico y de unas mínimas garantías de continuidad en la aplicación del mismo en el futuro, así como la opción de dar a conocer el proyecto y sus resultados en el ámbito internacional, permitiendo establecer una línea de investigación con un soporte de *Microsoft*.

13.4 ADECUACIÓN DE SSCLI A LOS REQUISITOS PLANTEADOS

Una vez escogida la implementación del estándar *CLI*, matizaremos cómo se adapta esta implementación concreta a los requisitos planteados en esta tesis.

13.4.1 Requisitos Arquitectónicos

En general todo lo dicho para el *CLI* sigue siendo válido para *SSCLI,* aunque se pueden hacer algunas matizaciones:

- **Máquina virtual**: La máquina virtual de *SSCLI* posee la misma arquitectura que el *CLR*, lo que le confiere una gran potencia y un funcionamiento correcto sin *bugs* importantes. Dicha máquina es completamente modificable y, de hecho, algunas de las partes del sistema han sido sustituidas por otras versiones que permiten facilitar las labores de modificación, haciéndolas menos complejas. Legalmente la licencia *Shared Source* no incorpora ninguna traba para hacer cualquier modificación a la misma. Además, incorpora un compilador *JIT* plenamente funcional adecuado para nuestros propósitos.
- **Independencia del lenguaje de programación**: En lo referente a *SSCLI*, los lenguajes soportados son *C#* y *JScript*, aunque es posible incorporarle más lenguajes y de hecho existen proyectos en marcha para ello, dado que *SSCLI* no ha eliminado esta posibilidad.
- **Independencia del problema:** Idéntico a *CLI*.
- **Soporte para capacidades reflectivas:** Idéntico a *CLI*.
- **Portabilidad**: La versión de *SSCLI* utilizada es soportada en plataformas *Windows, FreeBSD* y *Mac OS X*. La labor de portabilidad se ve facilitada gracias a la existencia de la *PAL* anteriormente mencionada.
- **Interoperabilidad:** Idéntico a *CLI*.
- **Extensibilidad**: *SSCLI* tiene un amplio soporte para su extensión con nuevas capacidades, ya que está provisto de:

- Acceso completo a la librería estándar, disponiendo de todo el código que la forma, y herramientas para incorporar las modificaciones al sistema de forma eficiente.
- Todos los ficheros fuente de la máquina virtual y del entorno de ejecución completo están disponibles para su modificación, siendo posible ampliar el sistema añadiendo nuevas clases, funcionalidades y comportamiento. También se cuenta en este caso con herramientas que incorporan estos cambios de forma efectiva, reduciendo el riesgo afectar a otras partes del sistema y disminuyendo el tiempo necesario para construir un sistema extendido. Esto también incluye la *PAL*, (*Platform Access Layer*), lo que permite modificar la interacción con la plataforma nativa sobre la que se ejecuta el sistema, y el *JIT*, que nos permite modificar cómo se traduce el código intermedio a código del sistema nativo sobre el que se está ejecutando *SSCLI*.
- Acceso a compiladores, analizadores, intérpretes y especificación de lenguajes existentes en la máquina.
- No hay que olvidarse de la estabilidad de la versión disponible, de forma que no haya que migrar el código de la modificación a versiones nuevas muy frecuentemente, como se dijo anteriormente.

- **Proyección del sistema para el desarrollo de aplicaciones reales**: Si bien *SSCLI* no es en sí una plataforma específicamente destinada al desarrollo de aplicaciones comerciales, debe tenerse en cuenta que actualmente la distribución *CLR* de *Microsoft*, base de su producto *Visual Studio .NET*, es una de las más usadas del mercado y sin duda la más adecuada en este aspecto. Ante la imposibilidad de modificar directamente la misma, *SSCLI* es un buen punto de partida que nos puede permitir elaborar modificaciones para ella de manera indirecta, ya que ambos sistemas comparten un elevado grado de similitud en su arquitectura y muchos módulos comunes, al poder considerarse *SSCLI* como un subconjunto del *CLR*. Además, el hecho de que esta tesis haya recibido apoyo económico explícito e interés por parte de la propia *Microsoft,* abre la posibilidad real de trasladar el trabajo desarrollado a dicha plataforma comercial.

13.4.2 Requisitos de Rendimiento

Si bien, como se ha visto anteriormente, *SSCLI* no parece que sea el sistema globalmente más eficiente, debido a la ineficiencia del código generado por su *Jitter* [Vogels05], una mejora de este módulo podría mejorar sustancialmente el rendimiento del mismo, por ejemplo, incorporando el *Jitter* que posee la versión comercial (*CLR*). No obstante, el empleado ahora en *SSCLI* es más adecuado para nuestros propósitos, debido a que ha sido creado pensando en su modificación y este módulo, como hemos visto en el diseño del motor de ejecución, tendrá que ser alterado.

No hay que olvidar por último que usando *SSCLI* se deja abierta la posibilidad de portar la solución creada al entorno comercial de *Microsoft*, más eficiente. Actualmente, sólo *SSCLI* garantiza tener una arquitectura similar a *CLR* que permita hacer un traslado lo menos problemático posible de una a otra plataforma de las nuevas funciones desarrolladas.

13.5 DESCRIPCIÓN DE LA IMPLEMENTACIÓN DEL PROTOTIPO ЯROTOR

A continuación se hará un análisis de una serie de aspectos relativos a la implementación del prototipo sobre el sistema *SSCLI* seleccionado (denominado *ЯRotor*), sin entrar en excesivas consideraciones técnicas. Partiendo del diseño del modelo y del motor computacional ya descrito anteriormente, el propósito de esta sección es describir las decisiones más importantes de implementación tomadas y otras consideraciones que se han tenido en cuenta de cara a la implementación de un primer prototipo operativo, al que podrían seguir otros que afinen más ciertos subsistemas en el futuro. Para una descripción de *SSCLI* sin modificar se recomienda consultar el Apéndice A o [Stutz03], done podrá verse una visión mucho más técnica de todos los aspectos y subsistemas de la máquina, algunos de los cuales se modificarán para conseguir los fines descritos.

13.5.1 Introducción:

A la hora de definir la arquitectura y la forma en la que la construcción de este estándar de *CLI* ha sido planteada [Stutz03], se han hecho una serie de consideraciones generales que debemos tener en cuenta a la hora de decidir cómo implementar las operaciones reflectivas y sus optimizaciones según lo que se ha descrito anteriormente:

- Toda la arquitectura de la máquina está pensada para lograr el mejor rendimiento posible en tiempo de ejecución, comprometiendo cualquier otro aspecto.
- Por el motivo anterior, a la hora de representar todas las estructuras de datos en memoria se han tomado ciertas decisiones, como el modo de alinear los objetos en memoria, la forma de construir una serie de estructuras de datos o la existencia de ciertos datos en *offsets* de memoria concretos (ver Apéndice A). Este tipo de compromisos a la hora de situar objetos en memoria mejora la eficiencia, pero dificulta los cambios a dichas estructuras, ya que cualquier modificación de tamaño o relocalización de las mismas podría causar un mal funcionamiento de todo el sistema. El sistema original no contempla que la estructura y el tamaño de las instancias pueda modificarse en tiempo de ejecución, está optimizado agresivamente para la consulta de información en estas estructuras.
- La implementación del sistema base resulta poco flexible. En muchas puntos distintos de la implementación, por motivos de seguridad y eficiencia, el motor de ejecución comprueba el tipo, tamaño, composición y otros parámetros de las clases que se cargan y del código que se ejecuta, remarcando su orientación clara al soporte de lenguajes estáticos, pero dificultando también las modificaciones que necesiten una comprobación de tipos más flexible.

Estas consideraciones por tanto determinaran la forma de implementar las operaciones reflectivas.

13.5.2 Modificaciones a Alto Nivel

Siguiendo el diseño ya planteado, se entiende como servicios de alto nivel el conjunto de servicios que el sistema extendido ofrecerá a los programadores mediante los medios estándar que éste posee para el acceso a funcionalidades desde cualquier lenguaje de alto nivel. Estos servicios se ofrecerán como llamadas a métodos de la librería estándar *BCL*, ya que no deseamos modificar ni la sintaxis ni la semántica de ningún lenguaje de los que ofrece el sistema de partida. Por ello, a alto nivel, si el programador necesita acceder a los servicios que le proporcionarán capacidades de reflexión estructural, deberá hacerlo explícitamente, a través de llamadas a las primitivas que se proporcionaran a través de nuevas clases y operaciones integradas dentro de dicha librería. A alto nivel se ofrecerán todos los servicios descritos en el diseño del modelo computacional realizado en un capítulo anterior.

Dado que se ha establecido anteriormente la necesidad de integrar los nuevos servicios dentro de la infraestructura ya existente en la máquina, la mejor forma de hacerlo es crear un nuevo `namespace` en la librería e integrar allí toda la nueva funcionalidad en clases si es técnicamente posible. Este espacio de nombres se denominará `System.Reflection.Structural`, para indicar que proporcionará capacidades de reflexión estructural. Importando este paquete el usuario podrá acceder a la funcionalidad proporcionada por sus clases. Las clases ya existentes que se han modificado se mantienen en su misma localización anterior por motivos de compatibilidad. Otras clases han tenido que situarse en el `namespace System.Reflection` para no complicar demasiado el establecimiento de vínculos con otras clases auxiliares a las mismas.

13.5.3 Modificaciones a Bajo Nivel

La implementación a alto nivel descrita anteriormente permite acceder a cualquier servicio, pero su implementación estará integrada dentro del motor de la máquina en código *C/C++*, de manera que estos servicios realmente ejecutarán otros equivalentes que accederán a todas las estructuras de datos del motor con una mayor eficiencia. Según el diseño descrito, estos servicios de bajo nivel serán los que podrán invocarse de forma transparente por el código intermedio del sistema (*CIL*), sin que sea necesario una invocación explicita de los mismos. Por tanto, siguiendo el diseño del sistema, los mismos servicios reflectivos serán accesibles tanto desde la *BCL* como mediante las instrucciones del *CIL* preparadas explícitamente para ello, tal y como se especificará posteriormente.

En las siguientes secciones veremos cómo se han integrado completamente en la arquitectura del sistema dichos servicios, tanto en lo referente a datos (nueva información añadida) como a operaciones (implementación de las primitivas).

13.5.3.1 MANTENIMIENTO DE LA INFORMACIÓN REFLECTIVA

Idealmente, a la hora de añadir o eliminar información de un tipo o instancia cualquiera, lo más deseable sería que esa información nueva que se le introdujese a dicho tipo o instancia para describir los nuevos elementos (atributos o métodos) que ha adquirido vía reflexión estructural de forma dinámica en tiempo de ejecución, se integrase en la estructura del mismo, apareciendo de una forma cohesionada y coherente

con el resto de la información que el objeto posea después del proceso de compilación. De esta forma, la información nueva podría accederse usando los mismos procedimientos que el sistema ya posee, al no haber distinción entre ambos tipos de elementos. En *SSCLI* esto no ha sido posible, debido a la especial construcción y disposición en memoria de la información de tipos e instancias dentro de la máquina a la que antes se hacía referencia (ver Apéndice A). Por tanto, la información añadida a tipos o instancias debe alojarse en otra zona de memoria, de manera que para lograr una integración, en la medida de lo posible, de los nuevos miembros dentro de la estructura de su "propietario" es necesario realizar las modificaciones en el código nativo del núcleo del sistema, alterando la estructura interna de las clases para que refleje el contenido de la figura 13.4.

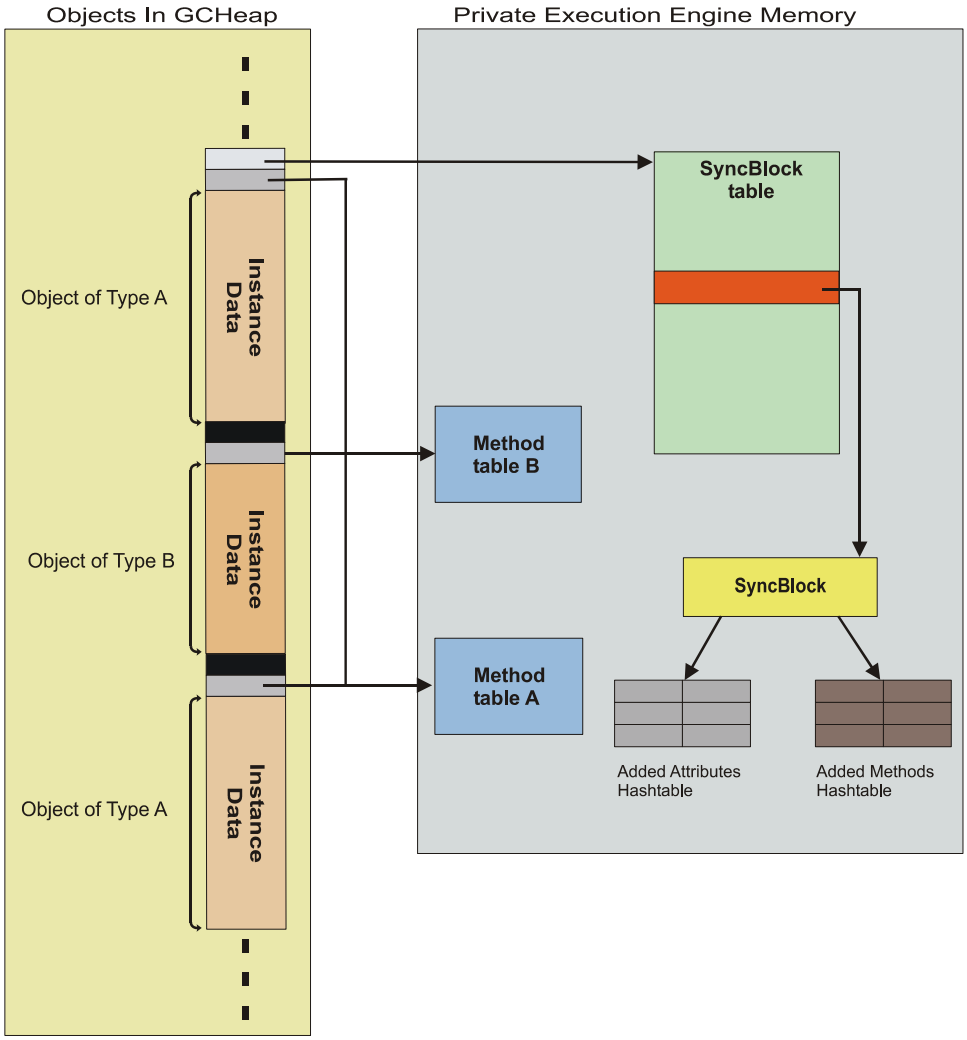

Figura 13.4. Estructura de soporte de nuevos miembros añadidos en *ЯRotor*

El principal problema que ha impedido la integración total de la información dinámica con la estática es que *SSCLI* no funciona correctamente si se hace una modificación del tamaño de una instancia una vez cargada en memoria. Para mejorar de esta forma el rendimiento general del sistema, se optimizan ciertos cálculos en el código para localizar miembros y manipular datos teniendo en cuenta esta circunstancia. Por ello, tal y como se ve en la figura 13.4, lo que se ha hecho es usar un elemento asociado a toda clase o instancia, denominado `SyncBlock`, y ampliarlo de manera que ahora contenga dos tablas *hash* donde se guardará la información relativa tanto a nuevos atributos como a nuevos métodos (una tabla para cada uno), aprovechándonos de esta

forma de las estructuras de datos existentes en *SSCLI* para guardar estos elementos. Por tanto, si un objeto o clase no tuviese elemento añadidos, no ocuparía más memoria de la que ocupaba originalmente. La otra aproximación estudiada se basaba en incrementar el tamaño de los objetos en cantidades fijas de espacio para N miembros, de manera que pudiesen alojar un determinado número adicional de elementos en un momento dado, y este espacio fuese adaptándose dinámicamente a la cantidad de miembros poseídos por una instancia o tipo incrementando o disminuyendo su tamaño de N en N.

Por tanto, cumpliendo con las premisas establecidas anteriormente, todo objeto y clase tiene asociado un elemento que permite guardar toda la información dinámica adicional de manera individual, pudiéndose acceder fácilmente al mismo a partir de una instancia dada. Para completar la capacidad de guardar nueva información, se ha tenido que usar también una estructura de datos especial de *SSCLI* denominada tabla de manejadores global (*global handle table*), donde se puede guardar la información de todos los miembros sin peligro de que el recolector de basura integrado en el sistema la elimine mientras está en uso.

La solución presentada es la que, de todas las estudiadas, permite guardar toda la información necesaria logrando un mayor nivel de integración sin causar efectos laterales al resto de módulos del sistema, soportando todas las funcionalidades requeridas y permitiendo representar al mismo tiempo modelos de herencia basados en concatenación (lenguajes estáticos) y basados en delegación (lenguajes dinámicos), de manera que si se ejecuta un lenguaje estático la información se obtenga de la estructura original de *SSCLI*, mientras que un lenguaje reflectivo pueda además consultar la información guardada en el `SyncBlock`.

Para guardar toda la información de un atributo añadido dinámicamente se usará la clase `RuntimeStructuralFieldInfo`, derivada de la clase `ReflectBaseObject` y que es una versión personalizada de la clase `RuntimeFieldInfo` ya existente a la que se le ha añadido información adicional como un nombre directamente utilizable para comparaciones (en tiempo de *debug* el nombre de los atributos permanece en una forma optimizada, pero difícil de manipular), propietario (para poder cambiarlo/reasignarlo fácilmente), tipo, valor (ambos para poder consultarlos de forma sencilla) y otros parámetros auxiliares para su correcta manipulación dinámica, así como métodos también necesarios para dicha tarea. Concretamente la clase es la siguiente:

```
//CHECKED refs used in the implementation when dealing with this class. We created the type
here.
#ifdef USE_CHECKED_OBJECTREFS
typedef REF<RuntimeStructuralFieldInfo> RSFIREF;
#else
typedef RuntimeStructuralFieldInfo *RSFIREF;
#endif

/*This class is the unmanaged part of the C# class with the same name. It contains a field
representation of the class in order to use its values into the execution environment with
ease, using them in order to communicate with the managed part. It also contains some
operations which were left to the unmanaged part in order to access them easily from other
modules.*/
class RuntimeStructuralFieldInfo: public ReflectBaseObject
{
public:
        StringObject *_name;
        Object *_owner;
        TypeHandle *_type;
        Object *_value;
        INT32 valueSelect;
        INT32 _attributes;
        INT8 _isDeleted;
        INT8 _isClass;
        INT8 _isOverload;

/* REDONDO: This method clone a runtime field, which is used when we have to "export" a class
added field to an object, using the lazy mechanism previously described.*/
        RuntimeStructuralFieldInfo *Clone ();
```

```
/* This method is used to test if the field is compatible with the specified one.
Compatibility is achieved if both fields have the same name and comatible (casteable) values.
This methods is used when we try to add a field and a deleted one is found.*/
        int IsCompatibleWith (RuntimeStructuralFieldInfo *f);

/* This method create a RuntimeStructuralField info who wraps a coded field (represented by a
FieldDesc class). Is used when we need to alter or delete a coded field, because we cannot
physically delete them due to the structure of the Rotor environment.*/
        static RuntimeStructuralFieldInfo *WrapFieldDesc (FieldDesc *f);
};
```

También puede apreciarse el método usado para clonarse, operación que se usará para el mecanismo perezoso ya descrito. En cuanto a los métodos, se creará una clase *MethodWrap* que usaremos para contener la información de los métodos añadidos a las clases. Esta clase, siguiendo el diseño hecho, contendrá parámetros como el propietario del método (para cambiarlo/reasignarlo), su signatura completa (de forma unificada, facilitando su uso en comparaciones), la referencia al código del método a partir del cual se extrae su código (según el procedimiento explicado en el apartado de diseño) y otra información adicional para su manipulación dinámica. La clase es la siguiente:

```
class MethodWrap;

#ifdef USE_CHECKED_OBJECTREFS
typedef REF<MethodWrap> MWRAPREF;
#else
typedef MethodWrap* MWRAPREF;
#endif

class MethodWrap: public Object
{
public:
        Object *Owner;

/*Al generar los objetos, aqui se debera guardar el toString del MethodInfo del metodo
generado.*/
        StringObject *Signature;

        //Sólo usado en caso de enmnascaramiento de metodos nativos.
        Object *_mInfo;

        INT32 _IsDeleted;
};
```

Tanto esta clase como su equivalente para atributos tienen una clase equivalente definida en un lenguaje de alto nivel, para poder usar objetos de este tipo en cualquier lenguaje de la plataforma y poder pasar y devolver estas estructuras de datos como parámetros o tipo de retorno de los métodos que se comuniquen con funcionalidades de bajo nivel. El sistema "traduce" la instancia de alto nivel en una de bajo nivel automáticamente ya que le hemos indicado esta equivalencia de manera que al construirse el sistema cree los medios de conversión adecuados por nosotros.

13.5.3.2 Semántica Extendida de la Máquina Virtual

Tal y como se especificó en el diseño del sistema, se ha extendido la semántica de ciertos *opcodes* del lenguaje *CIL* para que hagan uso de los nuevos servicios reflectivos integrados en la máquina. La intención es que cualquier acceso a miembros contemple la existencia de una posible información introducida en el sistema de forma dinámica y actúe en consecuencia, comportándose pues la ejecución de acuerdo con el modelo computacional descrito anteriormente.

Dado que operaciones como añadir o eliminar atributos o métodos de clases e instancias requerirían a priori la creación de nueva sintaxis en el lenguaje *CIL* para poder usarse (este lenguaje no posee ninguna instrucción que pueda servir o modificarse para ello), y no deseamos modificar este lenguaje, por el momento, en este prototipo inicial, sólo las operaciones de acceso a miembros/invocación de métodos podrán usarse de esta forma. Por tanto, cada vez que se haga un acceso a un atributo (lectura o escritura) o método (invocación), internamente el sistema deberá buscar, a partir de la información de la instancia o tipo sobre la que se haga el acceso, miembros añadidos que concuerden con los parámetros de la operación, devolviendo la información adecuada en cada caso en función de la operación realizada y el estado actual del sistema, siguiendo el diseño expuesto anteriormente.

Cada instrucción *CIL* genera un determinado código nativo, y es este último el que debe ser alterado para lograr la funcionalidad descrita como hemos visto antes. Para modificar el código nativo, se ha modificado la salida generada por el compilador *JIT* del sistema adecuadamente. Para ello ha sido necesario hacer dos operaciones principales:

- Se ha cambiado la forma en la que se realiza el chequeo de tipos del sistema, para desplazar parte de esas comprobaciones a tiempo de ejecución, ya que en otro caso no sería posible obtener el grado de flexibilidad que requieren nuestras operaciones reflectivas. El *JIT* ha sido alterado para no realizar ciertos chequeos en tiempo de compilación, de manera que en el caso de que se detecte un acceso a un miembro cuya información no está disponible en las estructuras de datos a las que tiene acceso el *JIT* en ese momento (se debe tener en cuenta que los miembros nuevos se añaden dinámicamente en la propia ejecución del programa, por lo que el compilador *JIT* no tendrá acceso a ellos), en lugar de devolver un error se generará un miembro "falso", que haga las veces del miembro buscado, y que permita al proceso de compilación seguir su curso normal como si el miembro al que accede existiese estáticamente. Por supuesto, la generación de este miembro falso necesita una determinada información para ser construido, información que se obtendrá de la propia instrucción que genera el acceso (de donde sacaremos el nombre, tipo y propietario).

  Posteriormente, a través del código creado por nuestras modificaciones, todos los accesos a miembros serán interceptados por el motor del sistema extendido, de forma que se puedan hacer las comprobaciones de tipo necesarias además de implementar la funcionalidad reflectiva correspondiente. Debe tenerse en cuenta que en cualquier caso es esta función de intercepción la que tiene la ultima palabra en cuanto a decidir si el acceso a un miembro es o no válido (existe, existe con otro tipo, se ha borrado, etc.), y que el *JIT* se ha modificado básicamente para permitir que el acceso a miembros no existentes estáticamente no devuelva un error y aborte el proceso, permitiendo pues la implementación de flexibilidad dinámica.
- Normalmente, el *JIT* del sistema generaría un código que traduce cualquier instrucción del *CIL* que implique un acceso a un miembro en una instrucción nativa que opera directamente sobre una dirección de memoria. Como hemos visto en el diseño del sistema, este tipo de accesos directos ya no sirven para nuestros fines y se han cambiado por la llamada a una serie de métodos propios, que permiten explorar la información disponible en tiempo de ejecución y tomar las decisiones correctas de cara a lograr el modelo reflectivo ya descrito. La figura 13.5 muestra el esquema de cómo estos accesos a miembros son interceptados, de acuerdo con el diseño realizado anteriormente. Ahora el código nativo generará, por cada acceso que el *JIT* evalúe como susceptible de ser interceptado, el código necesario para invocar a una rutina del motor de ejecución añadida también como parte de las modificaciones llevadas a cabo que a su vez se encargará de coordinar la llamada a nuestros servicios reflectivos, hacer las comprobaciones y optimizaciones adecuadas y devolver el valor que permita al código seguir su

ejecución normalmente, tal y como se hacia en el sistema original.

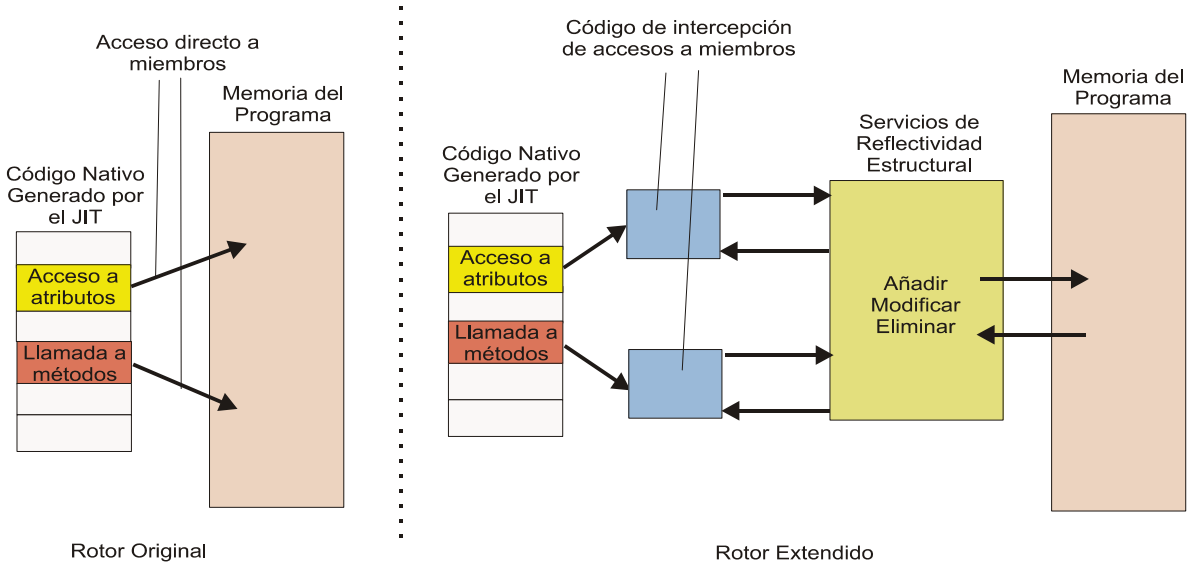

Figura 13.5. Esquema de implementación de operaciones reflectivas

La modificación del *JIT* del sistema es compleja, ya que el código nativo generado debe ser modificado para llamar a funciones que controlen cada acceso a miembros que se haga en el mismo, de manera que se puedan interceptar estos accesos e implementar en las funciones el modelo computacional adecuado. Alterar el código nativo que genera el *JIT* exige usar las facilidades de generación de código que el sistema proporciona, de forma que se puedan generar nuevas instrucciones nativas que ejecuten la llamada a dichas funciones en lugar del acceso al miembro que se realizaba anteriormente.

Las funciones creadas para el control de acceso a miembros se llaman *helpers* y se encargan de recoger la información en tiempo de ejecución proporcionada por el *JIT* (básicamente el miembro al que se intenta acceder) y la operación concreta a realizar, creando, a partir de dicha información, la secuencia de llamadas correcta para las primitivas reflectivas que intervengan en la operación en concreto, extendiendo de esta forma el modelo computacional de *SSCLI* para soportar funcionalidades de reflexión estructural. Para la adaptación del lenguaje *CIL*, se ha modificado la semántica de las instrucciones `ldfld`, `ldflda`, `ldsfld`, `stfld`, `stsfld` (acceso a miembros de instancia y de clase) `call` y `callvirt` (llamada a métodos).

Los *helpers* son un mecanismo que el sistema original ya poseía para acceder a miembros que provenían de operaciones remotas o similares, particularmente aquéllos que el sistema no es capaz de traducir a una referencia en memoria concreta. Este mecanismo ha sido extendido y modificado (creando nuevos *helpers* y modificando los existentes) para ser utilizado también en accesos "normales", empleando nuevamente herramientas del propio sistema para conseguir nuestros fines.

Más en detalle, la descripción de la funcionalidad poseída tanto por las instrucciones como por los *helpers* mencionados es la siguiente:

- ***ldfld***, ***ldsfld*** y ***ldflda***: Carga en la pila el valor del atributo (de instancia o estático) o su dirección, siguiendo el modelo computacional visto. De esta forma se garantizará el acceso a cualquier miembro (añadido o no) desde el *CIL*.
- ***stfld*** y ***stsfld***: Guarda un valor dentro de un atributo de instancia o estático, decidiendo en tiempo de ejecución cuál es la localización en memoria más propicia

para ello, independientemente de si el miembro ha sido añadido o no.

- ***call*** y ***callvirt***: Ejecutan métodos soportando estrategias de herencia basadas en concatenación y delegación, localizando el método adecuado en cada caso.

En lo relativo a *helpers*, las modificaciones realizadas para ello han sido:

- Modificar el código ensamblador que el compilador *JIT* genera dinámicamente para los métodos `compileCEE_{ LDFLD, LDFLDA, STFLD, CALL, CALLVIRT}` del mismo, correspondientes a las instrucciones vistas.
- El ensamblador generado fue modificado para generalizar el uso de estas funciones *helper*, realizando llamadas a las mismas en muchos más casos que el sistema original. Para ello el *JIT* separará los casos en los que pueda existir algún tipo de información reflectiva potencial de aquéllos que no, aunque tiene una información limitada para ello, y no podrá hacer una separación exacta de todos los casos, de lo que se encargará finalmente el propio *helper* en la ejecución del programa. Cuando se determine la posibilidad de que exista información reflectiva, el código generado se transformará en una llamada a uno de los *helpers* modificados, en función de la operación que se esté realizando (`JIT_{Set, Get}Field{32, 64, Obj}`, `JIT_GetFieldAddr` y `JIT_GetStaticFieldAddr`). Si el *JIT* determina que el acceso no va a necesitar ninguna operación reflectiva, entonces empleará el mecanismo del sistema original para acceder a la información.
- Siguiendo el mismo principio, se han creado dos *helpers* adicionales para obtener dinámicamente la dirección de llamada a un método concreto. Éstos son `JIT_Test{Method, VirtualMethod}`, cuya invocación tuvo que ser añadida al flujo de código nativo generado originalmente. Igualmente, si el *JIT* identifica que esa llamada no va a causar operaciones reflectivas, seguirá usando el mecanismo del sistema original para la llamada.

Finalmente cabe decir que será la propia implementación del *helper*, o en último caso la de la propia primitiva reflectiva la que determine si realmente hay un acceso posible a información dinámica o no, ya que estos elementos contarán con una información más completa que el propio *JIT*. En caso de que no se encuentre información reflectiva, se tratará siempre de recurrir a los mecanismos de acceso a la información que el sistema original poseía, para tratar de aprovechar su eficiencia.

El carácter interpretado del lenguaje *CIL* permite al programador crear programas en este lenguaje que accedan a miembros no existentes en tiempo de compilación, ya que su carácter interpretado hace que carezca de comprobaciones estáticas. Por tanto, actuando a nivel de *CIL* de la forma vista se logra que este lenguaje pueda usar todas las características reflectivas y además que nuevos lenguajes, cuyo sistema de tipos sea más flexible, puedan aprovecharse de estas características directamente generando código *CIL* que ahora sí podrá responder a sus necesidades de flexibilidad en vez de emularlas mediante una capa de código adicional. Esto abre la puerta a la implementación de lenguajes dinámicos o que usen un sistema de tipos mixto estático/dinámico que generen directamente código *CIL* y que por tanto se puedan ejecutar con una eficiencia mayor, al aprovecharse de todas las características que la máquina abstracta ofrece para procesar este código.

13.5.3.3 COMPORTAMIENTO DE LAS OPERACIONES REFLECTIVAS

Hasta ahora hemos visto cómo se pueden implementar los dos puntos de acceso a las primitivas reflectivas. Pasaremos ahora a describir cómo se han implementado las primitivas en cuestión [Redondo06b] [Redondo07]. Tal y como se especificó en el diseño, el conjunto de primitivas reflectivas implementadas estará en el motor de ejecución del sistema, integradas lo más posible dentro del mismo y usando todas aquellas estructuras de datos que se puedan reutilizar para los propósitos de esta tesis, así como otras facilidades de codificación, tipos y demás elementos que el sistema nos ofrezca y que podamos aprovechar.

A pesar de que se ha diseñado un modelo computacional híbrido que permite la "convivencia" de lenguajes estáticos y dinámicos, el principal problema que posee implementar el modelo teórico que rige el comportamiento de las operaciones reflectivas sobre el modelo de partida basado en clases, es el peligro que existe de alterar inadvertidamente este último de manera que se pierda la compatibilidad con código heredado. Por ello, cada primitiva debe ser tratada con extremo cuidado para respetar las reglas establecidas por el modelo computacional descrito sin que los cambios realizados alteren la máquina de forma no prevista. A continuación analizaremos algunas operaciones a implementar en las que se ha tenido que realizara alguna decisión de implementación relevante:

- **Obtener atributos y métodos (*get*{*Field*, *Method*})**: Esta operación plantea dos problemas principales que pueden existir a la hora de su implementación: La visibilidad y la accesibilidad.

    - La visibilidad de un atributo o método añadido debe respetar los convenios de visibilidad en programación orientada a objetos que existen en el *SSCLI*, formados por tres palabras reservadas: `public`, `protected`, y `private`. Por tanto, sería conveniente impedir el acceso a un atributo concreto si el elemento desde el que se trata de acceder al mismo no posee los privilegios necesarios para usarlo, de la misma forma que un objeto A no puede acceder a atributos privados de un objeto B si A no tiene vinculo con B que le dote específicamente de dicho privilegio. Esta funcionalidad no ha sido implementada en este primer prototipo, comportándose todos los miembros añadidos como miembros `public`.
    - En cuanto a la accesibilidad, se quiere englobar bajo este término el hecho de que, según el diseño descrito anteriormente, todo atributo añadido a una clase debe ser accesible por sus instancias como una copia privada (si no son estáticos) o no (si lo son). Para el primer caso, el proceso de copia de los atributos añadidos a una clase para que formen parte también de sus instancias (evolución de esquema) no es una labor trivial y necesita ser estudiada para una correcta implementación. Si se trata de acceder a un atributo de una instancia, y éste se detecta como uno añadido perteneciente a su clase, la operación de obtener el atributo debe hacer lo necesario para dejar al sistema en un estado correcto según el modelo descrito. Para ello creará un duplicado del atributo de la clase a través de la mencionada clase `RuntimeStructuralFieldInfo` y se añadirá el duplicado a la instancia en cuestión cuando trate de hacer uso del mismo (el mecanismo perezoso especificado en el diseño). Nótese que este problema no existe con los métodos pertenecientes a la clase, ya que las instancias no necesitan poseer copia individual de los mismos.

Para que los accesos a atributos o miembros se hagan de forma correcta, el sistema extendido está preparado para dar siempre más prioridad a la información añadida a un objeto sobre aquélla que tenga definida en su clase, de manera que toda alteración que se realice en tiempo de ejecución pueda ser correctamente contemplada en el momento en el que se realice. De esta forma, a la hora de acceder a un miembro siempre se consultará primero la información añadida al mismo, y se podrán considerar siempre primero los cambios realizados en tiempo de ejecución sobre la información añadida estáticamente.

- **Añadir atributos o métodos (*add*{*Field*, *Method*}):** Para que la implementación sea coherente con el modelo computacional teórico desarrollado, las operaciones de adición de miembros se comportarán como sigue:

    - Añadir miembros a clases: El añadir miembros a las clases provocaría una evolución de esquema perezosa por motivos de rendimiento, como ya se vio anteriormente. Si se añaden nuevos métodos no es necesario hacer más comprobaciones, ya que los métodos pertenecerán a la clase (*trait object*) y son directamente utilizables por los objetos. Los valores iniciales que tendrán los atributos añadidos en todas las instancias que los adquieran mediante el mecanismo perezoso implementado serán por defecto, en función de su tipo, siguiendo el convenido de valores al respecto que el sistema establece.
    - Añadir elementos a objetos: Hacer cambios a la estructura de una sola instancia con el modelo computacional descrito para el sistema extendido no plantea incidencias. Dado que los objetos seran capaces de agrupar tanto atributos como metodos, podremos usar el espacio reservado para nuevos miembros que cada instancia tendrá (figura 13.4) para guardar cualesquiera miembros que el usuario desee añadirle, sin preocuparnos de romper la coherencia del modelo.

- **Modificación de atributos y métodos (*alter*{*Field*, *Method*})**: Estas primitivas se han implementado finalmente como una combinación de primitivas de adición y borrado de los elementos.
- **Eliminar atributos o métodos (*remove*{*Field*, *Method*})**: según el diseño del sistema, debe ser posible quitar cualquier atributo o método de la clase o tipo que sea propietario del mismo. Este diseño además especificaba que todo acceso a código que ya no existe se solucionaría si el sistema es capaz de detectar esta incidencia y devolver una excepción capturable que indique el problema exacto, de manera que el código del programa pueda reaccionar a esta situación sin originar un error crítico. La implementación ha seguido fielmente estos parámetros establecidos en el diseño. Otro problema es impedir que un elemento intente eliminar atributos o métodos sobre los que no tenga permiso, es decir, que no le pertenezcan. En concreto, no sería admisible que desde una instancia se eliminase un atributo o método de una clase, de forma que afecte a todas las instancias de dicha clase. Por suerte, gracias a que es posible obtener el propietario de cualquier miembro (ver descripción de clases contenedoras de información de miembros), esto es fácilmente implementable.

  Por otra parte, dada la imposibilidad de eliminar físicamente aquellos atributos definidos estáticamente por programa, debido a que no podemos alterar el tamaño y la estructura de una instancia, la eliminación de los mismos se realizará introduciendo un atributo nuevo con la misma signatura, pero que figure como eliminado (atributo que aparece en la clase `RuntimeStructuralFieldInfo` mostrada). De esta forma, como el sistema siempre da prioridad a la información añadida, será capaz de detectar esta circunstancia, ya que al buscar un atributo de

nombre X, siempre encontrará primero el atributo añadido X con la marca de borrado activada, y podrá por tanto lanzar la excepción correspondiente.

- **Llamada a métodos (*`invoke`*)**: Como hemos visto en el diseño del sistema, para crear métodos susceptibles de ser añadidos a clases e instancias de la forma más eficiente posible, todo método destinado a este fin deberá estar creado previamente. La implementación de este mecanismo sigue fielmente lo dicho en el diseño, empleándose la mencionada clase *`MethodWrap`* para guardar la información de los métodos añadidos e invocarlos sobre cualquier objeto sobre el que el método se añada. Se ha tenido que modificar la comprobación del propietario del método que se hace dentro del motor del sistema, cambiándola por una que permita una mayor flexibilidad y por tanto hacer la operación descrita.

En lo relativo a parámetros e información necesaria por cada primitiva en el momento de su llamada, se establecen los siguientes convenios:

- Las primitivas {*`add`*, *`remove`*, *`alter`*, *`get`*, *`exist`*}*`Method`* reciben un objeto o clase (*`System.Type`*) como primer parámetro, para indicar si la operación asociada se hará sobre un objeto o sobre el comportamiento compartido por los mismos (*trait object*). El segundo parámetro es un objeto *`MethodInfo`* del espacio de nombres *`System.Reflection`*, ya existente en la máquina sin modificar. Este objeto describe completamente el método en cuestión, sus parámetros, su tipo de retorno, atributos y modificadores, es decir, permite identificar cada método de forma única. Mediante el uso de la propiedad *`IsStatic`* de este segundo parámetro se puede seleccionar si el método añadido se comportará como estático o no.
- La primitiva *`invoke`* ejecutará un método sobre un objeto o clase especificando su nombre, tipo de retorno y parámetros (su signatura completa, que permitirá identificarlo correctamente frente a otros métodos similares).
- Las primitivas {*`add`*, *`remove`*, *`alter`*, *`get`*, *`exist`*}*`Field`* pueden modificar la estructura en tiempo de ejecución de instancias o de su esquema común (clases o *traits*), según lo que se pase como primer parámetro, al igual que su rutina para métodos equivalente. El segundo parámetro es una instancia de la clase *`RuntimeStructuralFieldInfo`* vista, mediante la cual se describirá el tipo, visibilidad y el resto del estado del atributo en cuestión. El *flag* *`IsStatic`* también existe con los mismos propósitos ya descritos, de manera que determinará si añadir ese atributo a una clase producirá una evolución de esquemas o no.

Para ilustrar el conjunto de funcionalidades añadidas a la máquina, se ha creado este diagrama que muestra la disposición de todos los módulos añadidos al sistema (en color oscuro) que actúan conjuntamente con aquéllos que ya posee pero que han sido modificados (en color claro), siguiendo el diseño del sistema ya visto, las clases descritas y el resto de las consideraciones expuestas en este capítulo.

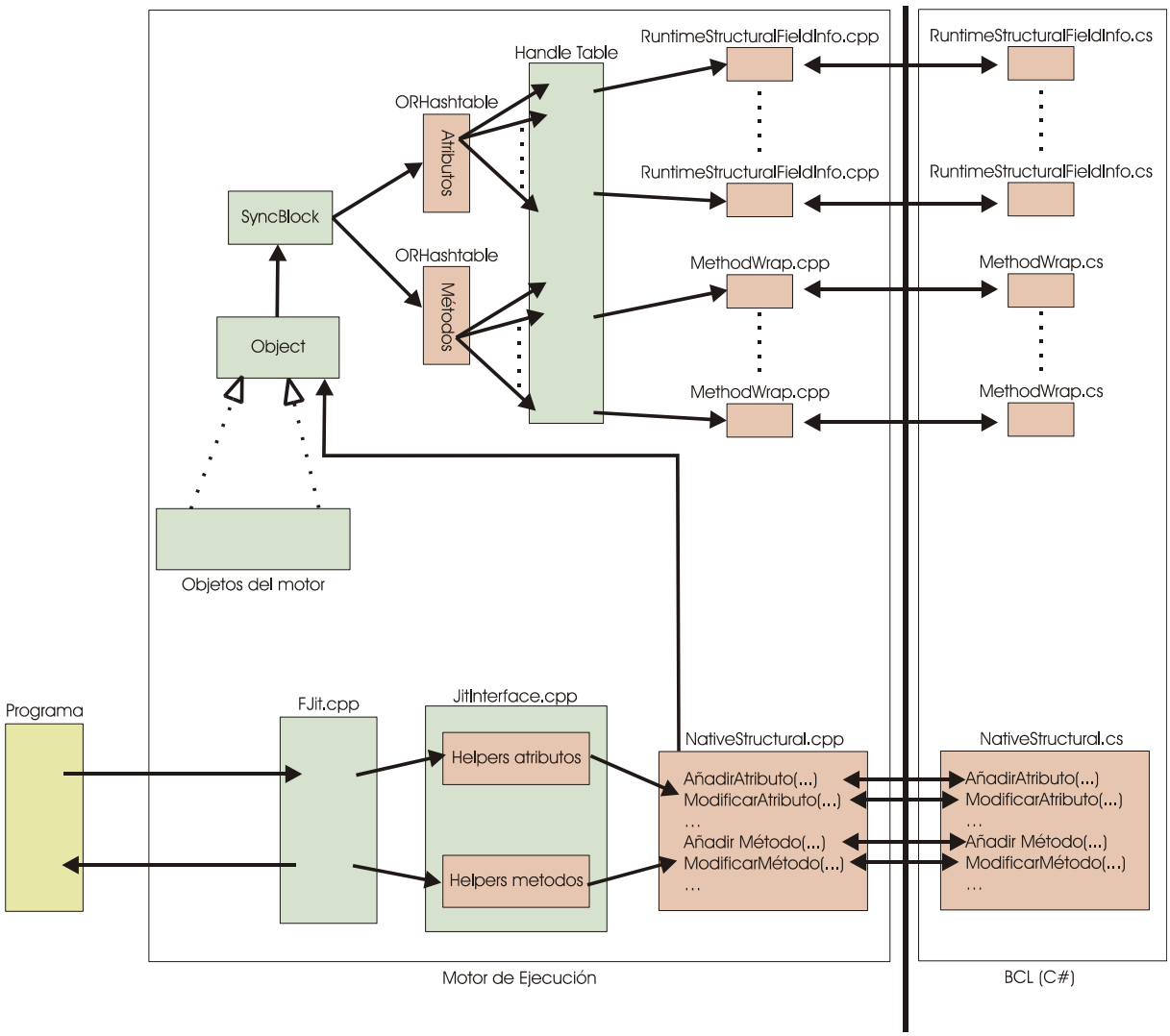

Figura 13.6: Estructura general de la implementación del prototipo

Como puede verse, en este esquema aparecen todas las clases que hemos mencionado en el diseño e implementación, su disposición en la arquitectura de la máquina y la relación existente entre ellas. Sólo mencionar la clase `ORHashtable`, que se ha creado como una tabla *hash* de funcionalidad mínima específica para guardar la colección de atributos o miembros de un objeto o clase dado, y que representa a las tablas *hash* asociadas al `SyncBlock` que se mostraban en la figura 13.4, y también la clase `NativeStructural`, que es sobre la que se centralizan todas las primitivas reflectivas construidas en la máquina. Por último, se recomienda que para una mejor comprensión de los elementos pertenecientes a la máquina mostrados en el esquema se recomienda consultar el apéndice A (construcción de *SSCLI* original y diseño de sus elementos principales) o el apéndice B (que muestra el conjunto de funciones de las clases creadas para el prototipo) o [Stutz03].

13.5.4 División de la Implementación

A la hora de abordar la implementación de las modificaciones, su gran complejidad aconsejó la división de dicha tarea en tres fases diferenciadas que se

describirán a continuación. Esta aproximación permite comprobar previamente las operaciones, su comportamiento y rendimiento a priori en un lenguaje de alto nivel, así como la viabilidad de la implementación y de ciertas formas de realizar algunas de las tareas necesarias, para después pasar a implementarlas a bajo nivel con unas mayores garantías. Por otra parte, abordar el desarrollo de esta forma es más sencillo al implementar las operaciones primero en un nivel en el que se cuenta con más facilidades de depuración, que permiten validar los conceptos teóricos expresados más fácilmente, y también permite tener diferentes versiones de las modificaciones para hacer estudios comparativos de rendimiento y optimizaciones. La implementación se ha dividido en tres pasos o *steps*:

- *Paso 1*: Todas las operaciones reflectivas descritas para el modelo computacional serán implementadas en uno de los lenguajes de alto nivel soportados por el *SSCLI*, escogiéndose finalmente *C#*. A pesar de las inherentes ventajas que esto posee, como la mayor facilidad de codificación, la gran cantidad de estructuras de datos que se pueden emplear para realizar las operaciones y el mejor control sobre errores, el rendimiento de las operaciones implementadas de esta forma es muy pobre, ya que todo el código se deberá traducir a *CIL* y éste a código ejecutable por la plataforma de destino, como un programa estándar. Este paso sería similar a la emulación de características reflectivas llevada a cabo por sistemas como *Jython* o *IronPython*.
- *Paso 2*: En este caso la interfaz de operaciones permanecerá visible al usuario como métodos de clases de alto nivel tal y como se diseñó para el *Paso 1*, pero la implementación de sus métodos se hará en *C/C++*, como parte de los servicios internos del núcleo de la máquina. Por tanto, cada operación descrita tendrá un equivalente a nivel interno, que se compilará junto con el resto del código del núcleo del sistema. A la hora de hacer la llamada al servicio, el sistema localizará el equivalente a bajo nivel ya compilado y lo ejecutará, siendo este código nativo mucho más rápido y eficiente que el anterior.

  Indudablemente, la implementación a nivel interno de todas las operaciones tiene ventajas. Una implementación dentro del núcleo de la máquina permitirá tener acceso a todas las estructuras de datos internas que mantienen los elementos de alto nivel. La estructura e información de control de tipos y clases, atributos, métodos, hilos, recolección de basura, etc. estará disponible ahora sin prácticamente limitaciones. Al poder acceder a mucha más información interna del sistema, se podrán escoger mejores alternativas para la implementación de ciertas características, así como manipular todas las estructuras que se consideren necesarias y optimizar mejor todas las operaciones. No obstante, la implementación dentro de la máquina es bastante más compleja que la hecha con un lenguaje de alto nivel, al contar con menos estructuras de datos reutilizables, servicios y sobre todo facilidades de depuración que permitan identificar problemas y errores fácilmente. Al producirse estos errores dentro del sistema, y no ser notificados de una forma tan explícita como los producidos por los lenguajes de alto nivel que soporta el mismo, identificar una fuente de error es una labor más compleja.
- *Paso 3*: En este último paso es el encargado de efectuar la modificación del compilador *JIT* de la máquina para que ciertas instrucciones del *CIL* usen los servicios reflectivos implementados dentro del sistema en el *Paso 2*, de manera que ahora se pueda acceder a los mismos transparentemente. Una vez finalizado, se logra la implementación del sistema completo tal y como se pretendía en el diseño original. Por otra parte, en esta fase se completarán también todas las optimizaciones al sistema que sea factible implementar para lograr el mejor rendimiento posible, una vez comprobadas las primitivas reflectivas introducidas y que se puede hacer uso de las mismas por cualquiera de las vías contempladas. Podemos considerar este *Paso 3* como la implementación final y válida del

prototipo siendo las anteriores pasos intermedios necesarios para llegar a ésta.

13.5.5 Optimizaciones

En esta sección de describirán brevemente las optimizaciones más relevantes realizadas en la implementación de *ЯRotor,* a partir de las expuestas en el capítulo correspondiente:

13.5.5.1 FJIT:

Como se dijo en el capítulo de optimizaciones, uno de los principales cuellos de botella del sistema extendido se origina cuando el compilador *JIT* transforma los accesos a miembros por llamadas a *helpers* que efectúan una serie de operaciones de búsqueda que permiten la implementación de capacidades reflectivas. Indudablemente, la llamada a estos métodos impondrá un coste de ejecución superior al acceso directo a una posición de memoria que se realizaba originalmente, por lo que interesa entonces que ese código de intercepción de los accesos esté lo más optimizado posible de cara a minimizar la pérdida de eficiencia en la que se va a incurrir. Para ello se han tomado las siguientes decisiones que optimicen el rendimiento de dichas llamadas, basadas en las optimizaciones ya descritas en el capítulo mencionado:

- En lugar de hacer una comprobación completa en el árbol de clases del propietario del miembro al que se esté accediendo en busca de información útil, se hará una comprobación reducida que simplemente determine si existe información añadida a alguna de estas clases o no. Esta segunda comprobación es más rápida que hacer una búsqueda completa, ya que se ha implementado comprobado simplemente si las tablas de miembros del `SyncBlock` existen o no y, dado que el número de potenciales objetos o clases con información añadida será comparativamente mucho menor a aquéllos que no la tengan, este proceso de chequeo previo tuvo un impacto muy positivo sobre el rendimiento del sistema final, ahorrándonos consultar todas las tablas.
- En el caso de los métodos, se ha hecho la optimización mencionada que se aprovecha del cálculo del *offset* que el compilador *JIT* hace para "saltar" directamente al código de un método si éste es estático. El sistema es capaz de calcular este *offset* mediante una operación muy rápida y sencilla, y este cálculo de dirección se aprovecha por los *helpers* para introducirla directamente en el código generado en caso de que no se haya encontrado ninguna información añadida dinámicamente. Ésta es una alternativa mucho más eficiente a realizar búsquedas entre los metadatos de los miembros de objetos y clases, comprobándose también un impacto muy positivo en el rendimiento.

13.5.5.2 MOTOR DE EJECUCIÓN:

Además de la disposición de la información añadida dentro de las instancias siguiendo los criterios de funcionalidad y optimización vistos, también podemos mencionar cómo se han optimizado las funciones de intercepción de accesos a miembros ya descritas anteriormente de acuerdo a las premisas enunciadas en ese capítulo. Para

ello, se ha estudiado el comportamiento de la máquina extendida en función de si se le ha añadido información dinámica o no. Si dicha máquina no posee información dinámica añadida en un instante dado, el acceso a los miembros se interceptará, pero se podrá retornar de cada llamada devolviendo el valor adecuado en el menor tiempo posible, sin hacer ningún tipo de búsqueda o comprobación, usando los mecanismos del sistema original. Esto se ha hecho mediante la optimización de "anotación" de clases mencionada, en su versión más sencilla (mediante *flags* globales al sistema de clases, separados para atributos y métodos, que indiquen si se ha añadido dinámicamente alguno de estos miembros hasta el momento), pero se espera que en una ampliación futura esta técnica de optimización pueda hacerse evolucionar a su versión más elaborada y lograr un aumento de rendimiento significativo. Este caso obtendrá por tanto la mayor eficiencia posible en tiempo de ejecución cuando el sistema no haga uso de las primitivas reflectivas, ya que el coste impuesto por las modificaciones será el mínimo posible en estas circunstancias.

13.6 SOPORTE PARA MODIFICACIONES A SSCLI: RFP

Por último, mencionaremos en esta sección el soporte dado por *Microsoft* a la elaboración de un prototipo que demuestre la idea planteada en esta tesis. *Microsoft Research* ofrece soporte para proyectos de investigación de *SSCLI* de propósitos diversos, mediante la creación de los *Request For Proposals* (*RFP*) [MSRRFP06] que esta empresa ha venido haciendo durante varios años a nivel mundial. Este proyecto se ha financiado con fondos otorgados por *Microsoft Research* precisamente como parte de dichos programas *RFP*, ganando un premio internacional la propuesta presentada sobre la que se basa esta tesis [MSRRFP06b]. Este premio se obtuvo mediante la presentación de la propuesta completa al concurso, en competición directa con proyectos de todo el mundo tras ser sometido a examen por parte de *Microsoft Research*, lo que avala el trabajo que se ha realizado.

En la actualidad el proyecto se ha terminado y presentado convenientemente ante el simposio organizado por la sede norteamericana de la compañía y se ha empleado para la realización de nuevas propuestas basadas en el mismo para futuras ediciones de los *RFP*, ganando un premio adicional, tal y como será descrito en la sección de trabajo futuro. Como se puede apreciar también en [MSRRFP06b], este proyecto es uno más dentro de la lista de proyectos en marcha que tienen como base *SSCLI*, demostrando que este sistema es válido como punto de partida para investigaciones sobre diversos aspectos relativos a máquinas abstractas y lenguajes de programación, como precisamente se ha hecho en este trabajo.

SECCIÓN D: EVALUACIÓN, CONCLUSIONES Y TRABAJO FUTURO

14 EVALUACIÓN DEL SISTEMA

En este capítulo se van a mostrar los resultados de la medición del sistema extendido en varios escenarios, para comprobar que las modificaciones realizadas permiten obtener un rendimiento adecuado para los objetivos expresados en esta tesis. En este sentido es importante tener en cuenta tanto el rendimiento como el uso de memoria del sistema extendido (su eficiencia), para que los resultados en rendimiento no se hagan a base de una sobreutilización de memoria, es decir, que el consumo de memoria adicional sea justificable con las ganancias en rendimiento obtenidas. No obstante, dados los objetivos de esta tesis, es preciso decir que nuestro principal objetivo es lograr el mejor rendimiento posible, y en ese sentido han ido orientados todos los cambios, es decir, no se han tomado medidas especiales para intentar emplear el menor espacio en memoria adicional posible, aunque si se han tomado para evitar su uso innecesario.

Por otra parte, en este capítulo aparecen una gran cantidad de gráficos que comparan múltiples sistemas tanto en memoria como en rendimiento. Los datos a partir de los cuales se han elaborado estos gráficos están recogidos en uno de los apéndices de esta tesis mediante tablas, y a estas tablas se hará referencia a lo largo de las explicaciones que se darán en este capítulo.

14.1 SELECCIÓN DE ELEMENTOS PARA LAS PRUEBAS:

14.1.1 Lenguaje de Pruebas

El rendimiento del sistema final se comparará con el que ofrece otra plataforma que dé soporte a lenguajes dinámicos y que tenga el mismo ámbito de aplicación. De todos los lenguajes dinámicos vistos anteriormente, se ha seleccionado para ello al lenguaje *Python,* por ser el que posee un mayor rendimiento y ofrecer primitivas dinámicas de reflexión estructural. *Python* es un lenguaje de propósito general, usado comercialmente y que goza de un gran soporte, así como diferentes versiones que permiten comparar nuestro sistema con implementaciones de diferente tipo. También se ha determinado que su eficiencia es muy elevada y superior a la que ofrecen otros posibles candidatos potencialmente seleccionables [Neumann03] [Wrenholt05], lo que ha determinado su elección. En concreto, se pueden citar los siguientes casos:

- ***Python* vs. *Ruby***: Existen estudios, como los 18 *test* del *Computer Language Shootout Benchmark* [CLSB06], que prueban que *Python* es un lenguaje más rápido, midiendo el rendimiento de dos implementaciones de *Python* y *Ruby* de

características equivalentes. El resultado obtenido muestra que el sistema *Python* es 4,6 veces más rápido, requiriendo 85,29% más memoria.

- ***Python* vs. *Smalltalk***: *Python* es 1,35 veces más rápido que *GNU Smalltalk*, necesitando 12,3 veces más recursos de memoria.
- ***Python* vs. *Self***: Las últimas versiones disponibles de *Self* tienen carácter experimental y no tienen soporte completo para todos los sistemas más conocidos (en concreto *Windows*). Por otra parte, la última versión insiste más en su carácter experimental asegurando que el sistema "dista mucho de estar perfeccionado" [Spitz06], por lo que no hemos creído conveniente hacer una comparación de ambos sistemas al no tratarse el último de un sistema finalizado y plenamente estable (lo que probablemente indica un rendimiento inferior).

14.1.2 Tipos de Sistemas de Pruebas

Una vez seleccionado el lenguaje a emplear, se debe tener en cuenta que el rendimiento ofrecido por el mismo será función de las características de la implementación usada. Por tanto, se deben seleccionar implementaciones de distinto carácter que puedan utilizarse para medir su rendimiento de forma comparada con nuestro sistema ante diferentes planteamientos. Antes de seleccionar ninguna implementación de *Python*, se establecerá la restricción de que sólo se contemplarán aquellas implementaciones que estén completas, es decir, que soporten la totalidad del lenguaje de forma correcta, descartándose por tanto toda implementación en proceso de desarrollo y/o que carezca de partes del propio lenguaje. Sólo empleando un sistema completo se podrá tener una medida fiable del rendimiento que ofrece, un sistema incompleto podría ofrecer cifras de rendimiento no reales, ya que los módulos en desarrollo del mismo, una vez completos, podrían afectar a estas cifras por introducir una carga de procesamiento extra que las afectase negativamente.

Por tanto, de todas las implementaciones posibles, se seleccionarán algunas cuyo procesamiento se haga sobre un sistema basado también en una máquina abstracta. No obstante, existirán dos posibles opciones al respecto:

- **Capacidades reflectivas no emuladas**: Sistemas dentro de esta categoría pueden ser *CPython* [CPython06] y *ActivePython* [ActivePython06].
- **Capacidades reflectivas emuladas**: Estas implementaciones usan una máquina virtual existente y emulan sobre ella sus capacidades dinámicas. Sistemas dentro de esta categoría son *IronPython* [IronPython06] y *Jython* [Jython06].

Por tanto, teniendo en cuenta todo lo dicho, seleccionaremos las implementaciones y versiones que aparecen citadas a continuación para nuestro estudio de rendimiento y consumo de memoria:

- ***CPython* 2.4** (normalmente llamado simplemente *Python*): Esta implementación es probablemente la más extendida, desarrollada en *C*.
- ***ActivePython* 2.4.0**: Otra distribución interpretada de *Python*, de *ActiveState*, disponible en sistemas *Linux*, *Solaris* y *Windows*.
- ***Jython* 2.1** (también llamado *JPython*): Una implementación hecha 100% en *Java* de este lenguaje. Ha sido integrada completamente con la plataforma *Java* 2.

- ***IronPython* 1.0**: Una implementación de este lenguaje que genera código *CIL*, *bytecode* de la máquina virtual *.NET* o del proyecto *Mono*.

14.1.3 Clases de Pruebas

Para medir tanto el rendimiento como la eficiencia del sistema creado (al que se denominará para abreviar *ЯRotor*) se ha usado un conjunto de pruebas que permitan medir tanto el rendimiento en tiempo de ejecución como el consumo de memoria en diferentes escenarios (la eficiencia del sistema extendido, en definitiva). Por tanto, una vez seleccionados tanto el lenguaje con el que comparar nuestro sistema final como las implementaciones del mismo susceptibles de ser comparadas, se establecerá también a qué tipo de pruebas se someterán todos ellos para determinar su rendimiento y eficiencia y probar la validez de la idea postulada en esta tesis. Estas pruebas son:

- **Eficiencia de primitivas reflectivas**: Medidas que comprueben la ganancia de rendimiento de las primitivas dinámicas implementadas en el sistema extendido frente a las mismas primitivas empleadas en las diversas implementaciones de *Python*, así como las diferencias existentes en cuanto a consumo de memoria.
- **Eficiencia frente a programas estáticos**: Se someterá a ambos tipos de sistemas a ejecuciones de *benchmarks* reales para comprobar el rendimiento y consumo de memoria de ambos sistemas ante programas que no usan capacidades de reflexión. Aunque un sistema optimizado para programas estáticos en principio debería ofrecer un rendimiento superior a un sistema preparado para lenguajes dinámicos, se desea comprobar cuál es el impacto de las modificaciones introducidas cuando se ejecutan programas que no requieren su uso, entre otras cosas.
- **Coste de la flexibilidad**: Comparación del sistema original frente al sistema extendido, empleando programas de *benchmark* reales que no usen las primitivas de reflexión creadas, para calcular que penalización de rendimiento y consumo de memoria supone introducir estas primitivas en el sistema al ejecutar código no reflectivo.

Por tanto, el abanico de pruebas diseñado contempla todos los casos relevantes de estudio: Comportamiento del sistema ante código reflectivo, comportamiento del sistema ante código no reflectivo y coste de implementación de soporte reflectivo en el sistema, siempre comparándolo con aquellas plataformas más eficientes en cada caso.

14.2 MEDICIÓN DE PRIMITIVAS REFLECTIVAS

Como hemos visto, la principal ventaja de los lenguajes dinámicos es su capacidad de modelar *software* dinámico y adaptable gracias a la reflexión que incorporan. Por ello, es muy importante medir la eficiencia de sus capacidades dinámicas. Para ello, hemos medido el tiempo de ejecución de todas las primitivas descritas en el capítulo de diseño del sistema, usando para ello bucles de 10.000 iteraciones en el caso de atributos y 2.500 en el caso de métodos, eliminando cualquier tipo de operación de entrada/salida y

grafica durante la ejecución de las mediciones. Todos los *test* fueron llevados a cabo en una máquina *Pentium* IV 3,2 *Ghz HT* con 1 *Gb* de *RAM* con *Windows XP*. Cada primitiva se ha probado en nuestro sistema y en cada una de las distribuciones de *Python* vistas.

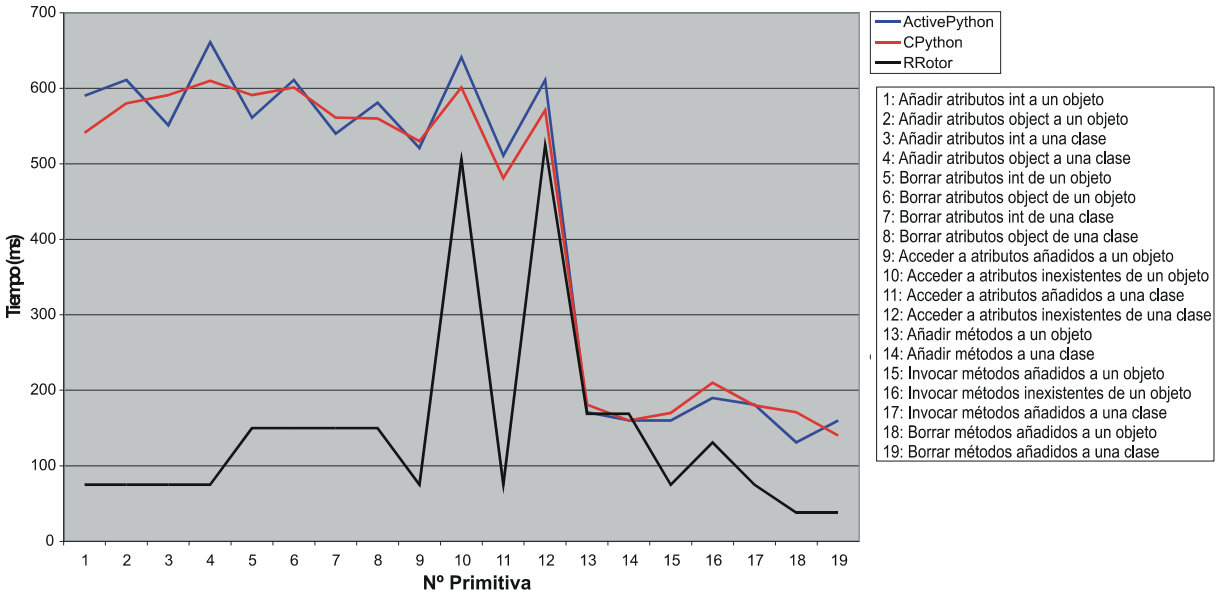

Figura 14.1: Gráfico de rendimiento de *ЯRotor* frente a *CPython* y *ActivePython*

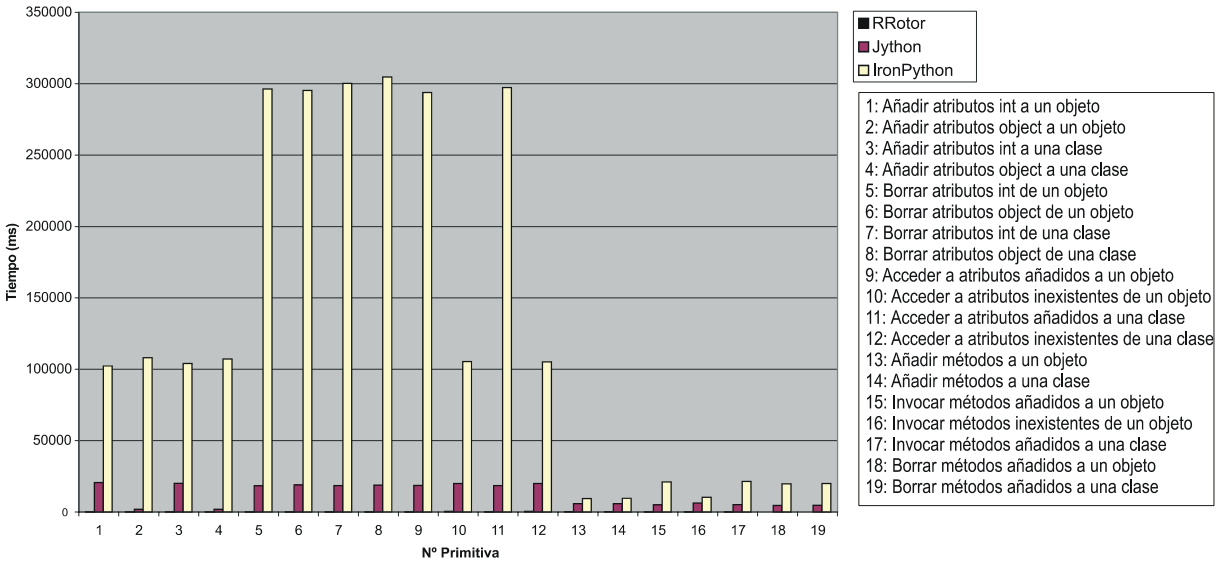

Figura 14.2: Gráfico de rendimiento de *ЯRotor* frente a *Jython* e *IronPython*

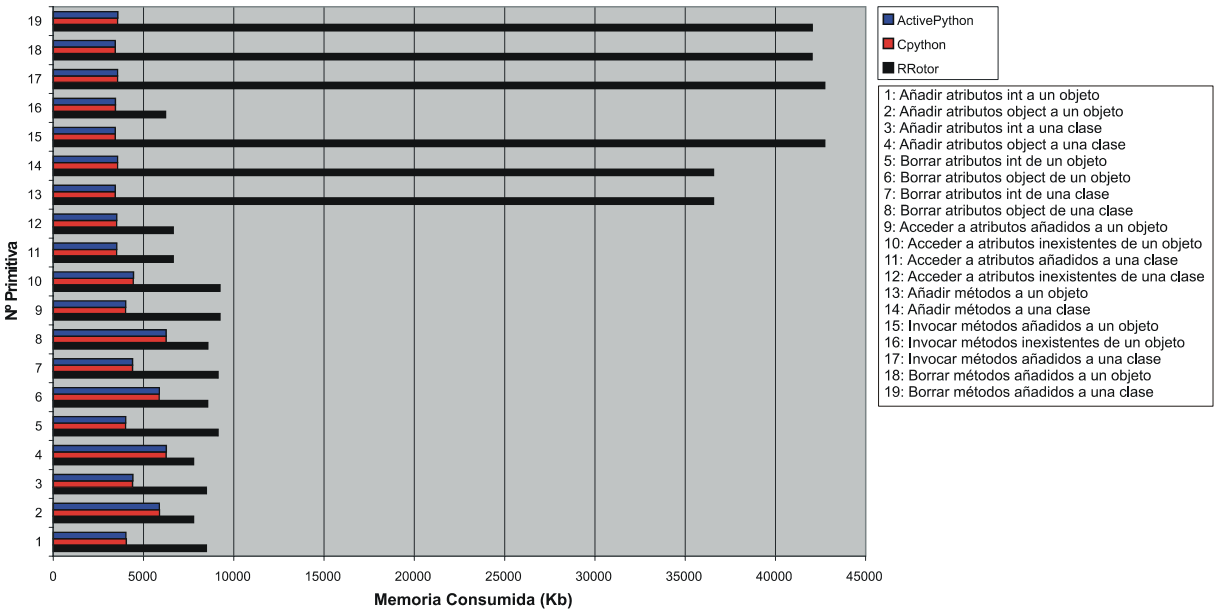

Figura 14.3: Grafico de uso de memoria de *ЯRotor* frente a *CPython* y *ActivePython*

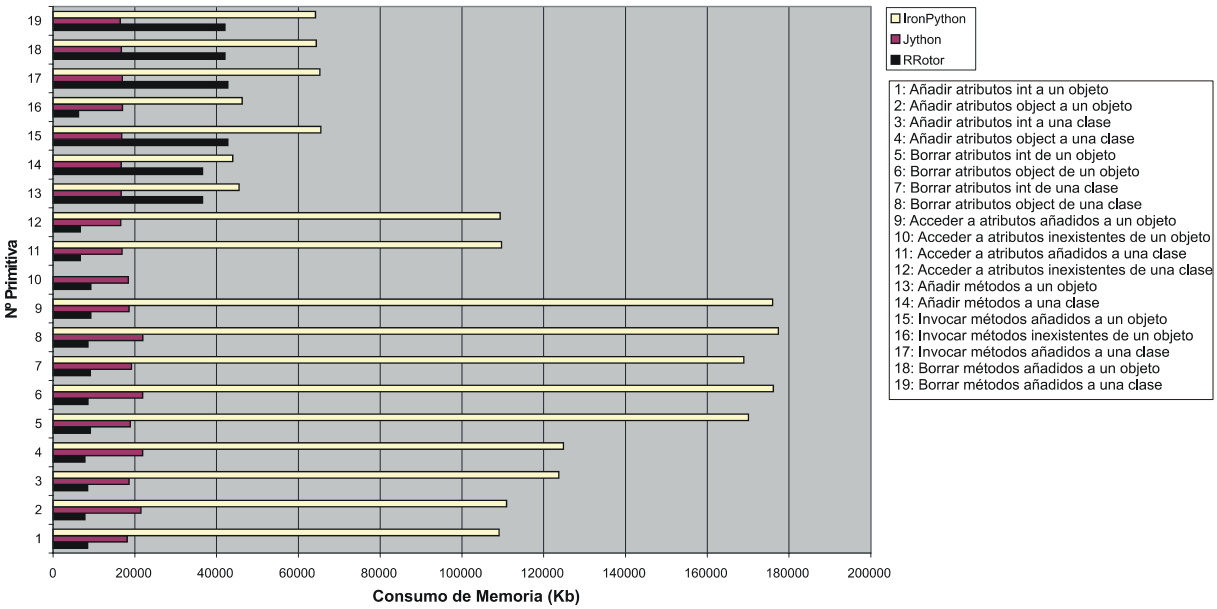

Figura 14.4: Grafico de uso de memoria de *ЯRotor* frente a *Jython* e *IronPython*

Los cuatro gráficos anteriores (correspondientes a las tablas 1 y 2 del anexo C) muestran el resultado de medir el tiempo que tardan en completarse las operaciones, cuyos datos están expresados en milisegundos, y los *Kbytes* de memoria necesarios para su ejecución. Como podemos apreciar, tanto *Jython* como *IronPython* tienen el peor rendimiento en todos los t*est*. La necesidad de implementar *Jython* como código 100% *Java* permite la interoperabilidad con cualquier programa *Java* existente, pero ocasiona una pérdida de rendimiento significativa, pudiendo aplicarse el mismo principio a *IronPython*. La generación de código *CIL* que simule el modelo de reflexión de *Python* sobre una plataforma que no lo soporta nativamente causa por tanto una pérdida importante de rendimiento en ejecución. El origen de está pérdida de rendimiento es probablemente el código adicional que estos sistemas deben generar para soportar el modelo de reflexión. Para estas pruebas, nuestro sistema fue compilado en el modo de

operación *free* (sin información de depuración y con el máximo nivel de optimización posible).

Por tanto, podemos apreciar cómo tanto *ЯRotor*, *CPython* y *ActivePython* ejecutan las primitivas de reflexión de forma más eficiente que los sistemas que generan código intermedio (*Jython* e *IronPython*). Si calculamos las veces que *ЯRotor* es más rápido que el sistema usado para la comparación de cada una de las primitivas analizadas, y hacemos la media aritmética del número de veces obtenido para todas ellas, el resultado es que nuestra plataforma es 113,24 veces más rápida que *Jython* y 1246,18 veces que *IronPython* respectivamente, datos obtenidos de los cálculos representados en la figura 14.5 y correspondientes a la tabla 3 del apéndice C.

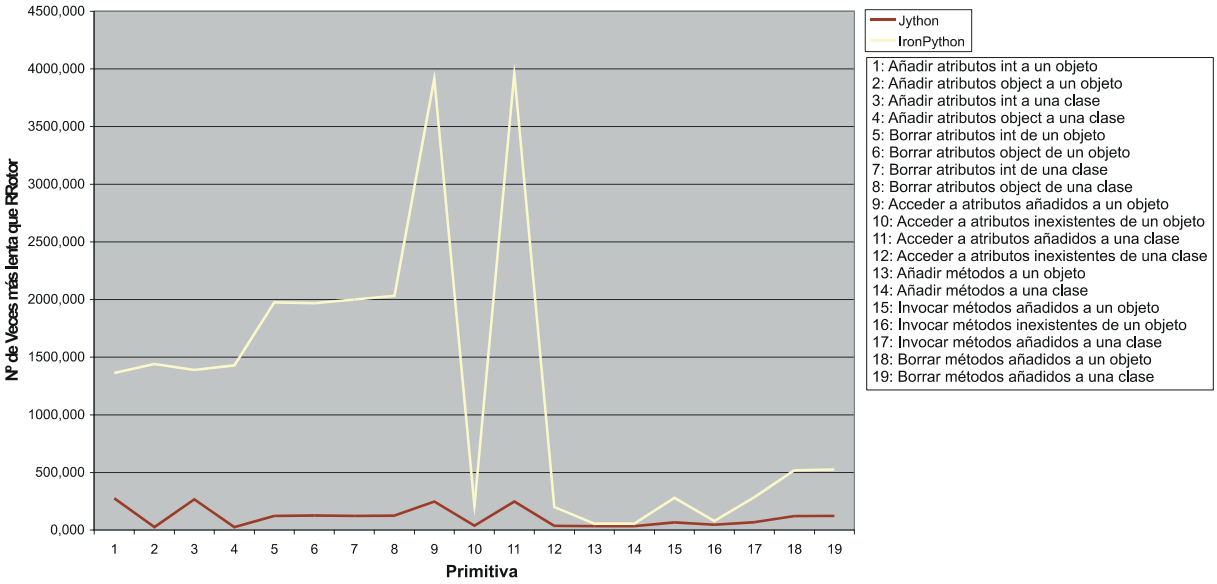

Figura 14.5: Rendimiento global comparado de *ЯRotor* frente a *Jython* e *IronPython*

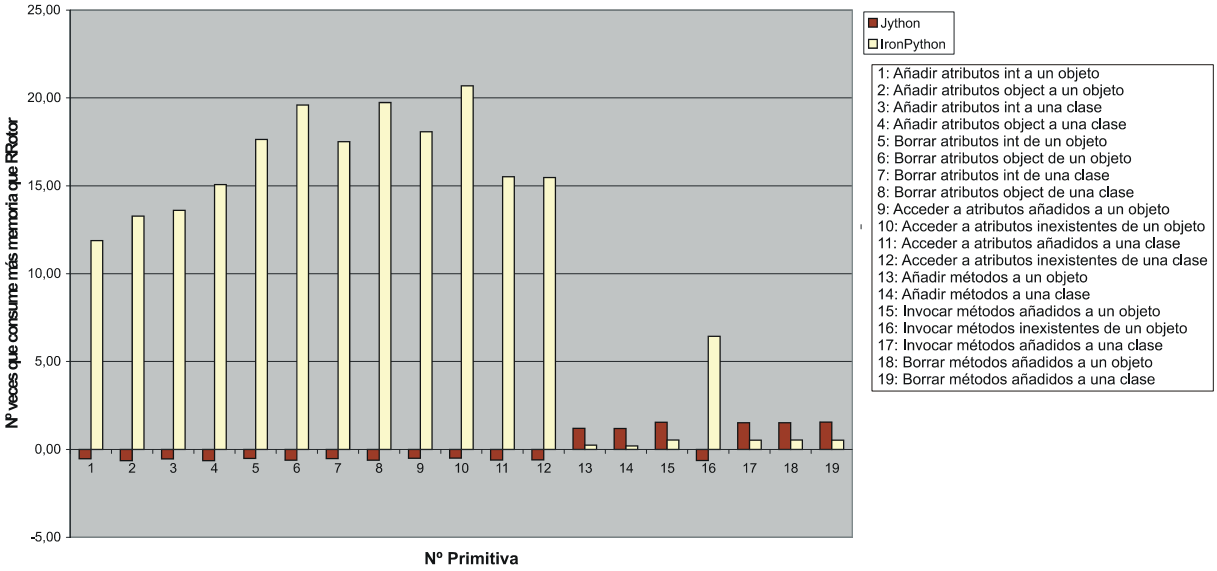

Figura 14.6: Uso de memoria global de *ЯRotor* frente a *Jython* e *IronPython*

En cuanto a consumo de memoria, aplicando el mismo procedimiento anterior, *ЯRotor* necesita un 6% más de memoria que *Jython*, pero *IronPython* usa 10,92 veces

más memoria que *ЯRotor*. La figura 14.6 muestra el número de veces que un sistema consume más que el otro por cada primitiva, representando los datos de la tabla 4 del apéndice C.

Efectuando los cálculos indicados anteriormente, finalmente obtenemos que *ЯRotor* es 3,17 veces más rápido que *ActivePython* y 3,14 veces más que *CPython*. *ЯRotor* necesita una media de 3,9 veces más memoria que los otros sistemas. Estos resultados se han calculado a partir de lo mostrado en las figuras 14.7 y 14.8, datos que también aparecen en las tablas 3 y 4 mencionadas anteriormente.

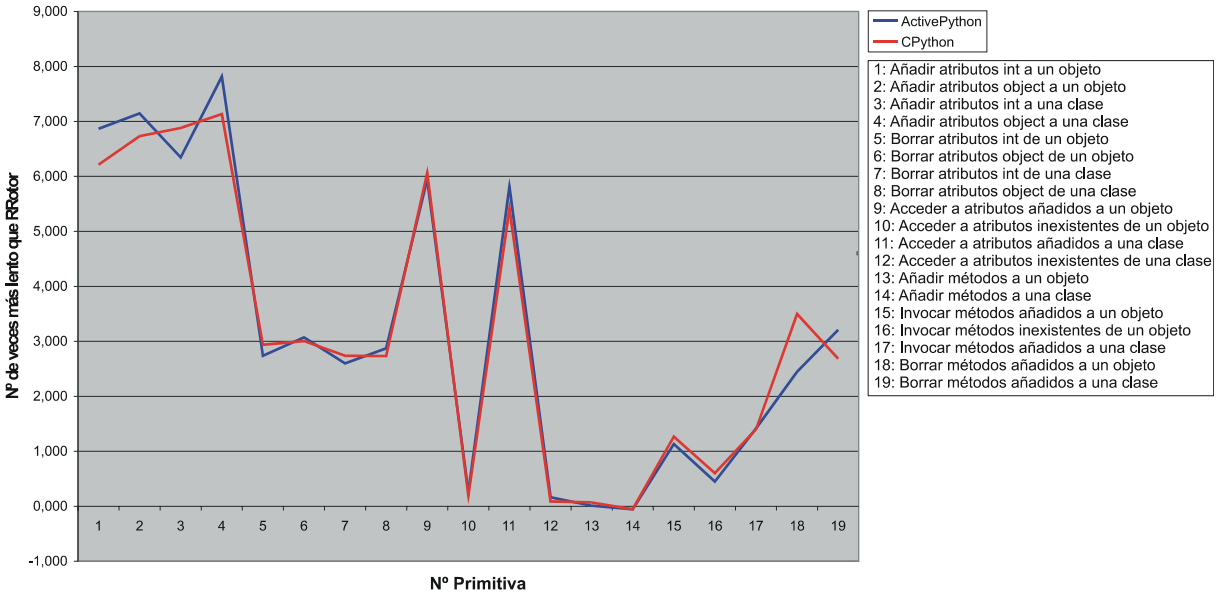

Figura 14.7: Rendimiento global de *ЯRotor* frente a *CPython* y *ActivePython*

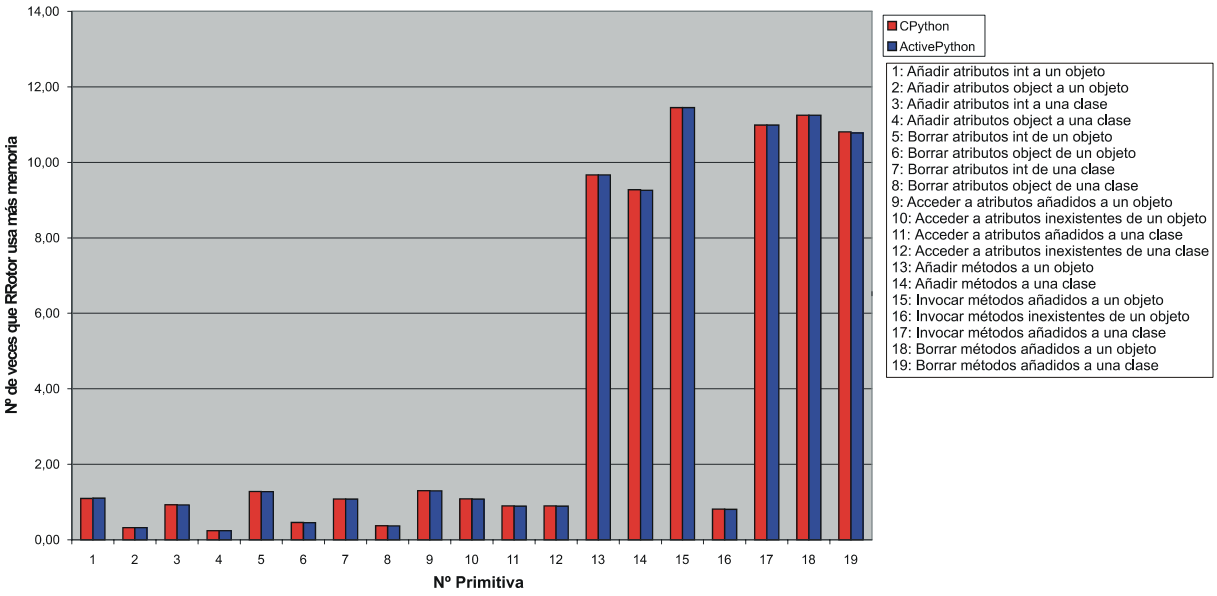

Figura 14.8: Uso de memoria global de *ЯRotor* frente a *CPython* y *ActivePython*

El único *test* en el que *ЯRotor* es un poco más lento es el consistente en añadir métodos a clases. La diferencia se atribuye al modo en que *SSCLI* trata los métodos. Según nuestra implementación, la única forma de acceder a los miembros de un objeto o clase es a través de la clase `OBJECTHANDLE`. Este mecanismo de indirección es el

empleado para implementar las referencias del recolector de basura generacional que el sistema posee, y en el caso de los métodos, el conjunto de operaciones implicado en el proceso de creación del nombre completo del método, acceder al mismo a partir de este *object handle*, y finalmente acceder a la implementación del mismo tiene un coste elevado al que se atribuye la caída de rendimiento.

Las otras tres primitivas que ofrecen una diferencia de rendimiento menor frente a las implementaciones de *Python* probadas son las que acceden a miembros inexistentes. Esto es debido al coste impuesto por el algoritmo de búsqueda de miembros que se diseñó anteriormente, cuya complejidad es elevada para permitir implementar la funcionalidad del modelo de herencia por delegación que usa el modelo computacional del sistema extendido, que requiere hacer un recorrido y una búsqueda por las clases "padre" de la dada antes de devolver un error.

14.3 MEDICIÓN DE RENDIMIENTO DE CÓDIGO NO REFLECTIVO

Aunque lenguajes "estáticos" como *C#* o *Java* serían un medio mucho más adecuado para desarrollar programas que no usen capacidades reflectivas, es bastante común que un programa realizado con un lenguaje dinámico, que use capacidades de reflexión, pueda tener porciones del mismo en las que no se use ninguna de esas capacidades dinámicas, comportándose como un programa estático. Para contemplar estos casos, en esta sección mediremos la eficiencia y rendimiento del sistema ante programas "estáticos", es decir, programas que no usen ninguna capacidad reflectiva.

Para hacer este tipo de medición se ha empleado el *benchmark Tommti*, diseñado por Thomas Bruckschelegel para evaluar las características de lenguajes como *Java*, *C#* y *C++* en *Windows* y *Linux*. Este *benchmark* está compuesto por 14 pruebas independientes que usan estructuras de datos básicas y operaciones aritméticas para efectuar mediciones de diversos aspectos clave de un lenguaje [Bruckschelegel05]. Las figuras 14.9, 14.10 y 14.11 muestran el resultado de ejecutar el *benchmark* tanto en *ЯRotor* como en *CPython* (la implementación más rápida de *Python* de todas las usadas), expresando el tiempo en milisegundos y la memoria en *Kb*. Para ello, el *benchmark* se ha traducido a *Python* sin alterar su diseño. Los gráficos siguientes corresponden a las tablas 5 y 6 del apéndice C y muestran la comparación en cuanto al rendimiento ofrecido y memoria consumida para cada primitiva de forma porcentual, según los valores devueltos por la ejecución de los *test*. Finalmente también se muestra un gráfico con la ganancia de rendimiento obtenida comparada con el consumo de memoria de cada una de las primitivas de las que consta el *test*, correspondiente con la tabla 7 del apéndice C.

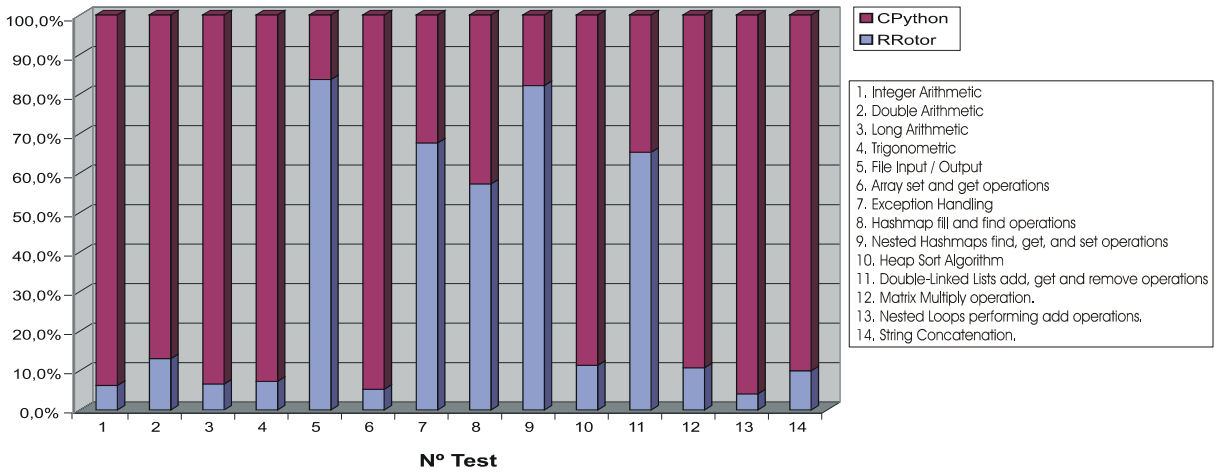

Figura 14.9: Rendimiento de código no reflectivo por cada test

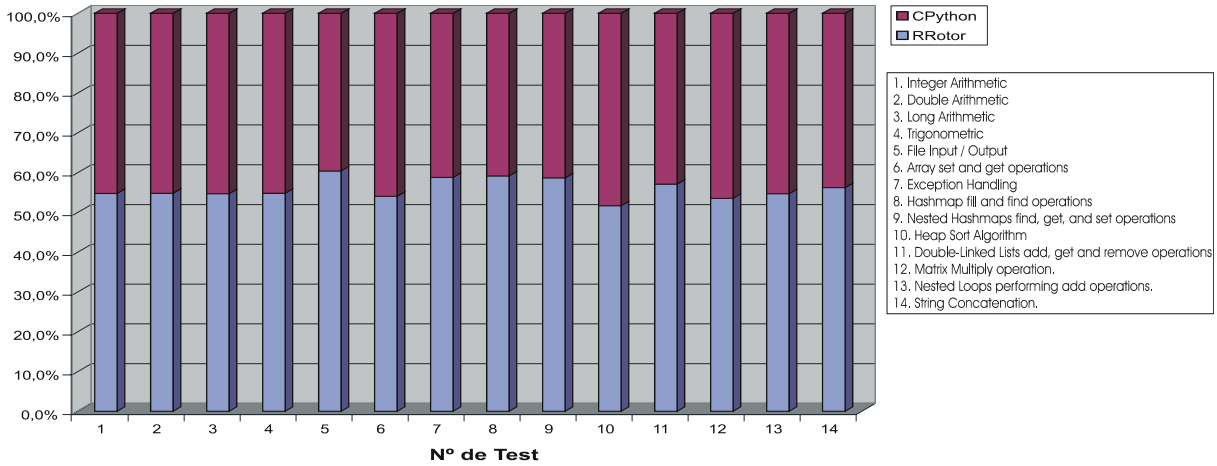

Figura 14.10: Uso de memoria de código no reflectivo por cada *test*

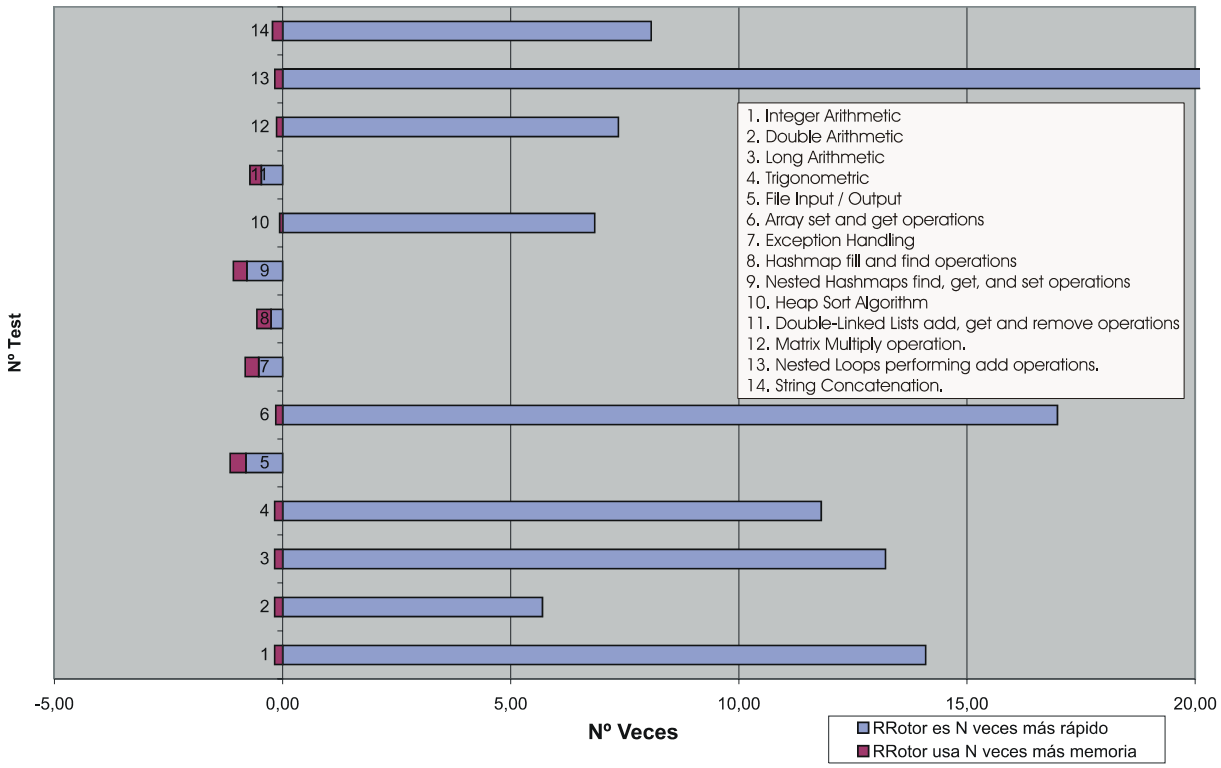

Figura 14.11: Uso de memoria relativo a la ganancia de rendimiento

Como se muestra en el gráfico anterior, hay dos clases de *test* concretos donde *CPython* es algo más rápido que *ЯRotor*. Respecto al primero de ellos, que englobaría operaciones básicas de entrada/salida de archivos y manipulación de miembros de clases contenedoras de la librería estándar, se atribuyen estas diferencias a que estos elementos contenedores manipulados se ofrecen dentro de la librería estándar *BCL* en el primer lenguaje, mientras que en *Python* pertenecen al propio intérprete, siendo el coste de acceso a los servicios de la *BCL* un factor que puede causar una importante pérdida de rendimiento y por tanto explicar estos resultados. En el caso de las excepciones (la segunda clase de *test* con inferior rendimiento de *ЯRotor*), tanto *SSCLI* como *CLR* implementan un mecanismo de manejo de excepciones con un rendimiento pobre [Bruckschelegel05].

Vemos pues como los programas que no usan capacidades de reflexión se ejecutan globalmente más rápido en *ЯRotor* que en *CPython (ЯRotor* es 7,41 veces más rápido que *CPython*, ver tablas 5 y 7 del apéndice C). Tal y como mencionamos anteriormente, los lenguajes dinámicos no son tan eficaces como los estáticos para este tipo de programas que no necesitan características de reflexión. Por ello podemos deducir que cuando un programador selecciona un lenguaje dinámico para implementar sus programas, es altamente probable que lo haga porque necesite la flexibilidad que le ofrece, por lo que es más importante medir el rendimiento de las primitivas reflectivas que del código no reflectivo. De todas formas, si nuestro sistema tiene un mejor rendimiento en ambos escenarios, sólo puede indicar que la aproximación tomada en esta tesis es realmente adecuada para soportar eficientemente bajo una misma máquina tanto lenguajes dinámicos como estáticos ofreciéndose el mismo tipo de soporte para ambos, estando de acuerdo con los requisitos planteados.

14.4 MEDICIÓN DEL COSTE DE INCORPORACIÓN DE MAYOR FLEXIBILIDAD AL SISTEMA ORIGINAL

La última variable a medir en nuestro estudio es el coste de incorporación de la flexibilidad al sistema original. La flexibilidad estructural en tiempo de ejecución es la principal característica ofrecida por los lenguajes dinámicos para la construcción de *software* adaptativo y adaptable, pero su implementación requiere un coste de memoria y de rendimiento en tiempo de ejecución respecto al código que no usa ninguna clase de reflexión. Para medir este coste se ha empleado el *benchmark Tommti* descrito en la sección anterior, comparando *ЯRotor* con el *SSCLI* sin modificar. Las figuras 14.12 y 14.13 muestran los resultados obtenidos, correspondiente a la tabla 3 del apéndice C.

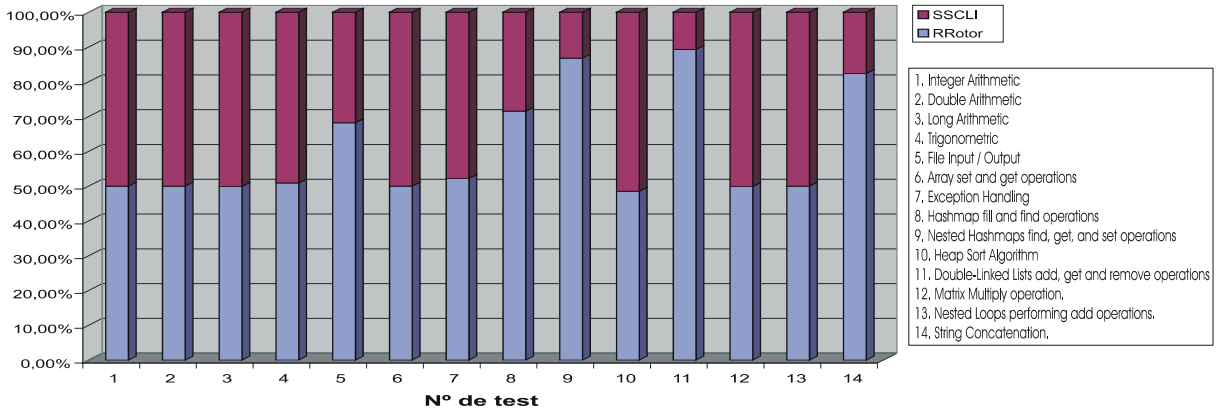

Figura 14.12: Coste en rendimiento de incorporar características de reflexión a *SSCLI*

El coste adicional de uso de memoria es un 7% (ver tabla 8, apéndice C) en todas las primitivas, mientras que la desviación típica de la penalización de rendimiento es de 164,36% (tabla 9, apéndice C). Esto es debido a que algunos *test* no tienen coste de rendimiento, como se ve en la figura 14.12, mientras que otros son significativamente más lentos. Estos *tests* más lentos son precisamente los que efectúan un gran número de accesos a miembros no reflectivos. Por ello, se ha evaluado el coste tanto de acceder a un atributo como de invocar a un método, obteniendo que en ambos casos la penalización de rendimiento es en torno al 60%, debido a los cálculos adicionales introducidos para soportar el modelo para lenguajes dinámicos que se ha introducido.

Por último, para obtener una estimación más aproximada del coste de rendimiento hemos ejecutado programas "estáticos" reales sobre *ЯRotor*, empleando para ello los *benchmarks LCSCBench* y *AHCBench* recopilados por Ben Zorn [Zorn06]:

- ***LCSCBench***: Está basado en el *frontend* de un compilador de *C#*, escrito en *C#* y usando un algoritmo generalizado *LR* (*GLR*) para efectuar el parseo. Este *benchmark* usa intensivamente tanto la *CPU* como la memoria, necesitando gran cantidad de memoria *heap* en función de los ejemplos usados. La ejecución de este *benchmark* por *ЯRotor* es 0.81 veces más lenta que la de *SSCLI*, necesitando 1.11% más memoria.
- ***AHCBench***: Este programa está basado en un algoritmo de compresión y

descompresión de archivos usando la compresión adaptativa *Huffman* [CProgrammingHuffman06]. *AHCBench* tiene 1267 líneas de código y usa intensivamente la *CPU* de la máquina, aunque no es tan intensivo en cuanto a memoria. *ЯRotor* necesita un 8% más de espacio en memoria y ejecuta el *benchmark* 2,14 veces más lento que el *SSCLI* original.

Las figuras 14.13 y 14.14 muestran los datos antes mencionados, mostrando los datos de tiempo en segundos y la memoria en *Kb* (tablas 10 y 11 del apéndice C):

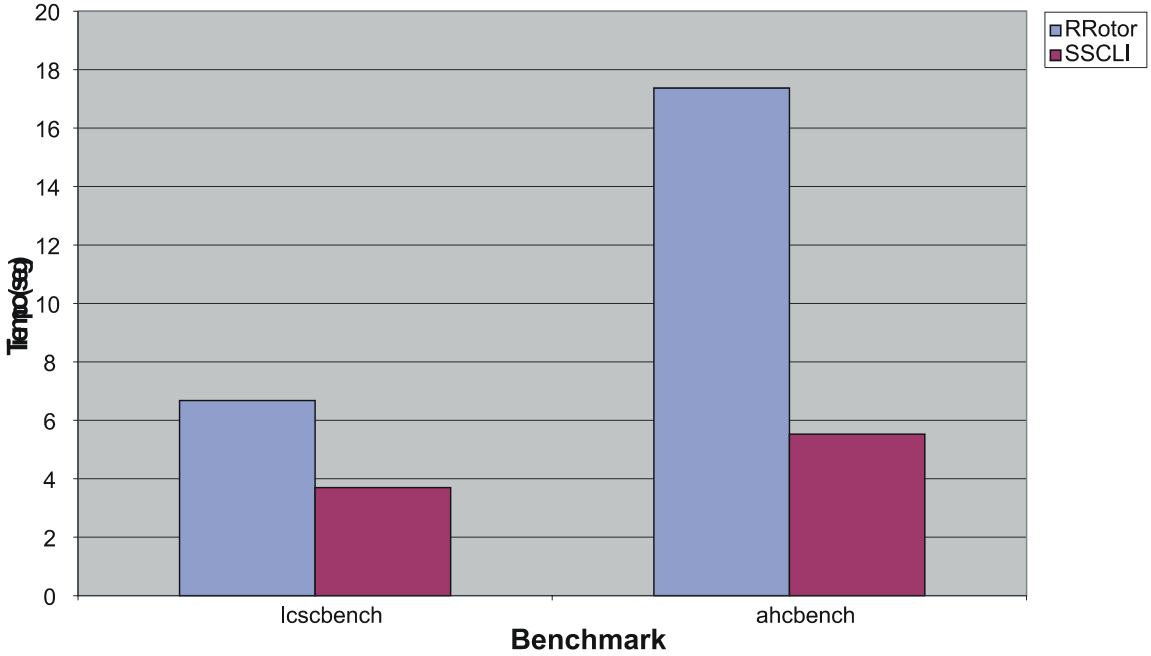

Figura 14.13: Rendimiento comparado de *lcsc* y *ach* en *ЯRotor* y *SSCLI*

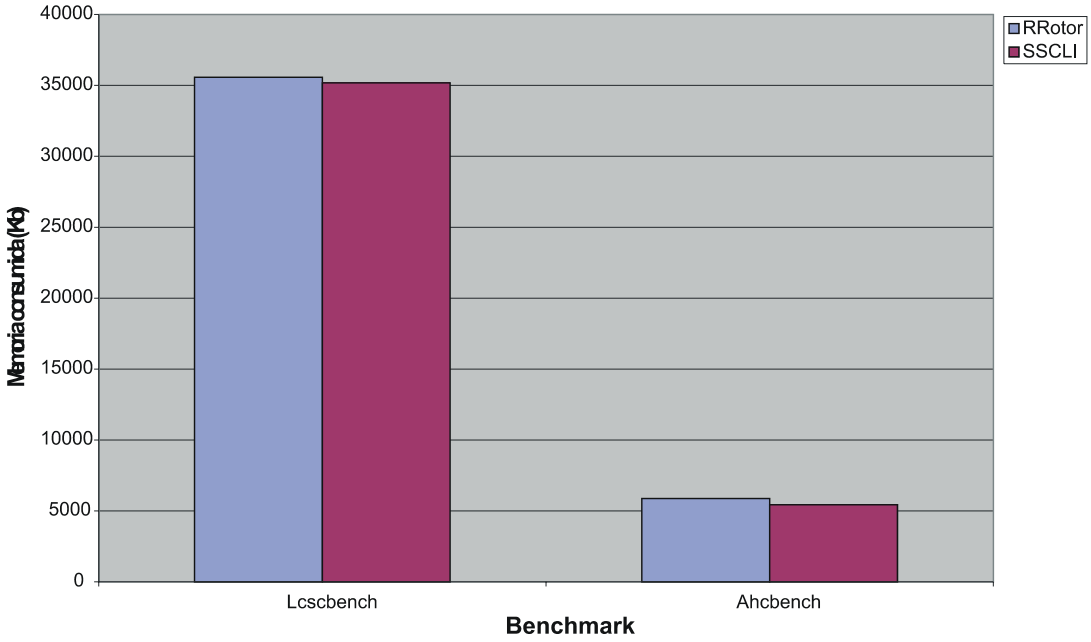

Figura 14.14: Uso de memoria comparado de *lcsc* y *ach* en *ЯRotor* y *SSCLI*

14.5 ADECUACIÓN DE LOS RESULTADOS A LOS REQUISITOS DE RENDIMIENTO

A continuación evaluaremos los resultados de rendimiento y eficiencia conseguidos respecto a los requisitos planteados al principio de este trabajo.

14.5.1 Rendimiento Frente a Sistemas de Características Similares

El sistema se ha mostrado globalmente más rápido frente a las implementaciones de *Python* con las que ha sido comparado, tanto en aplicaciones que usan reflexión (donde aquellos sistemas que emulan sus capacidades de reflexión, como *IronPython*, tienen un rendimiento notablemente peor que el resto) como en aquéllas que no la usan. En el primer caso es cierto que algunas primitivas ofrecen un rendimiento bastante mejor que otras, pero deben tenerse en cuenta que el sistema implementado es un prototipo, el cual es probable que pueda optimizarse más en un futuro, y que globalmente se alcanza un rendimiento 3 veces superior a la implementación más rápida de *Python* que se ha podido encontrar. Todo ello nos permite afirmar pues que nuestro sistema ofrece un rendimiento superior a los existentes en la actualidad y que por tanto la idea de partida de esta tesis se ha visto verificada.

En cuanto a los programas que no usan características reflectivas, nuestro sistema es más de 7 veces más rápido que *Python*, validando nuevamente la idea presentada y dejando claro como un lenguaje estático es más adecuado para las aplicaciones que no necesiten usar un determinado grado de flexibilidad que un lenguaje dinámico.

Por último, en cuanto al consumo de memoria, nuestro sistema consume más memoria que sus equivalentes dinámicos que no emulan sus características de reflexión (sobre todo a la hora de tratar con métodos), pero esto es algo necesario de cara a obtener mejoras apreciables de rendimiento y debe tenerse en cuenta que las plataformas de partida tienen un funcionamiento y una manipulación de la memoria que consumen muy diferente. Las estructuras de datos necesarias para la manipulación de métodos son las que más disparan este consumo, que no obstante es apreciablemente inferior al consumo que hacen otros sistemas implementados sobre máquinas virtuales de características de partida muy similares o idénticas a la que hemos usado en este trabajo, tal y como puede apreciarse en la tabla 4 del apéndice C.

En un análisis más en profundidad de estos datos de consumo de memoria frente al rendimiento que se obtiene con ellos, es interesante comprobar cómo en los sistemas de mejor rendimiento (*ActivePython* y *CPython*) el promedio de ganancia de rendimiento se sitúa en torno a 3 (nuestro sistema es de media unas 3 veces más rápido que éstos), mientras que el promedio de consumo de memoria adicional también se sitúa en torno a 3 (nuestro sistema consume de media 3 veces más memoria que éstos). Por tanto, la ganancia de rendimiento obtenida está a la par del consumo de memoria adicional necesario, no considerándose por tanto un consumo excesivo. Es probable no obstante que se puedan hacer optimizaciones en este aspecto en un futuro al prototipo diseñado para hacer las pruebas.

14.5.2 Penalización de Rendimiento Respecto al Sistema de Partida

En los estudios realizados se ha visto cómo efectivamente modificar la máquina virtual de partida para dotarla de más flexibilidad ha tenido un impacto apreciable en el rendimiento, aunque este impacto varia mucho en función del programa que se ejecute y las partes a las que acceda. Debido al modelo computacional implementado, los mayores costes se derivan del acceso a miembros de clases o instancias, debido al algoritmo de búsqueda de miembros que ha sido implementado. No obstante, el coste adicional no se ha disparado y queda dentro de unos límites bastante razonables, a lo que ha contribuido sin duda el gran número de optimizaciones introducidas al prototipo.

En cuanto al consumo de memoria, al ser programas que no usan las características de reflexión añadidas al sistema, éste no es elevado, consistiendo normalmente en que el sistema extendido tiene una huella en memoria algo superior al de partida (debido al espacio adicional que ocupan las modificaciones hechas) y no crece apreciablemente durante la ejecución (puesto que no se hace uso de ninguna estructura de datos que contenga miembros dinámicos). Por tanto, en este último caso se considera que el coste de memoria es adecuado.

En definitiva podemos afirmar que el coste derivado de la introducción de características de flexibilidad al sistema original es contenido y aceptable para las ventajas que hemos obtenido.

14.6 PERSPECTIVAS DE MEJORA

En el desarrollo de *ЯRotor* siempre se ha tenido especial cuidado en el diseño de las funcionalidades implementadas, para que el rendimiento ofrecido por las mismas fuese el máximo posible y no se viese comprometido de partida por un mal planteamiento de base. Los primeros resultados obtenidos por el sistema inicialmente construido [Ortin05b], aunque prometedores, creíamos que eran ampliamente mejorables, y por ello se ha empleado una considerable cantidad de tiempo en diseñar e implementar una serie de optimizaciones como las descritas en un capítulo anterior. De esta forma, tras la implementación y prueba de las optimizaciones descritas, hemos mejorado dichos resultados hasta llegar a los niveles expuestos, que validan nuestra tesis de partida. Por ejemplo, en el rendimiento de las primitivas de reflexión incorporadas, los resultados han pasado de ser ligeramente mejores que las implementaciones de *Python* más rápidas con las que hemos comparado el sistema hasta ser más de 3 veces más rápidos, logrando por tanto una ganancia de rendimiento notable.

En el caso concreto de la penalización de rendimiento respecto al sistema de partida, se ha conseguido pasar de 1,2 a 0,81 veces más lento en el caso del *benchmark lcsc* y de 3,71 a 2,14 veces más lento en el caso del *benchmark ahc*. Creemos que el desarrollo de más optimizaciones y el refinamiento de las actualmente existentes en un futuro probablemente mejorará este aspecto, lo que nos permite afirmar que estos resultados podrían mejorarse aún más.

15 APLICACIONES DEL SISTEMA DESARROLLADO

Siendo éste un trabajo dedicado a la obtención de un soporte más eficiente en tiempo de ejecución para los lenguajes dinámicos, y destinándose por tanto a tratar de solventar el que probablemente sea el principal problema que poseen este tipo de lenguajes (la eficiencia), este sistema podrá estar presente en todos aquellos sitios donde el uso de un lenguaje dinámico esté justificado, lenguajes que se implementarán sobre la máquina y que previsiblemente podrán ofrecer un mejor rendimiento para estas aplicaciones.

Por tanto, teniendo en cuenta este hecho, las potenciales aplicaciones del sistema pueden ser muy numerosas y variadas, pero no haremos una descripción pormenorizada en este capítulo. No obstante, sí que describiremos algunas de ellas para ilustrar tanto el posible uso futuro del sistema como la importancia que están teniendo los lenguajes dinámicos en la actualidad.

15.1 IMPLEMENTACIÓN Y USO COOPERATIVO DE LENGUAJES DINÁMICOS

Como ya se ha dicho, la implementación de lenguajes sobre máquinas virtuales tiene una serie de ventajas (portabilidad, interoperabilidad,…). No obstante, debe tenerse en cuenta sobre todo la característica de interoperabilidad que posee el sistema que hemos usado como base. Mediante esta característica, cualquier nuevo lenguaje que se implemente sobre él mismo podrá usar sin dificultad cualquier otra funcionalidad ya desarrollada en otro lenguaje del sistema, ventaja que debemos tener en consideración debido a la gran importancia que posee. De esta forma, la implementación de un lenguaje dinámico sobre nuestro sistema extendido podrá contar con una elevada serie de funcionalidades ya desarrolladas para la creación de sus programas, con lo que, por ejemplo, no tendríamos que desarrollar una librería estándar de clases para cada nuevo lenguaje creado.

Además, una de las aplicaciones más importantes se deriva de esta característica precisamente. Si logramos la "convivencia" bajo la misma máquina de lenguajes tanto estáticos como dinámicos, entonces podremos desarrollar aplicaciones que empleen diferentes tipos de lenguaje para hacer sus desarrollos, empleando aquel tipo que más se adapte a las necesidades del módulo *software* que se esté desarrollando. Por ello, podríamos desarrollar aplicaciones que usen un lenguaje estático en aquellas partes que requieran la máxima eficiencia y robustez, y que a su vez cuenten con partes desarrollados con lenguajes dinámicos en aquellos puntos donde sea necesario un alto grado de flexibilidad (como por ejemplo en módulos que necesiten implementar algoritmos de inteligencia artificial), todo ello pudiendo cooperar ambas clases de código sin encontrarse con ninguna barrera para ello.

Una máquina como la elegida, que cuente con soporte para ambos tipos de

lenguajes y que no limite cuantos lenguajes se pueden implementar sobre la misma, permitirá un desarrollo de aplicaciones más rico y potente, al permitir al usuario seleccionar cuando quiere hacer uso de la flexibilidad y permitirle incluso seleccionar cómo (ya que podría elegir cualquier lenguaje soportado por la máquina), todo ello con un coste de ejecución inferior al que los lenguajes dinámicos de capacidades similares pueden actualmente ofrecer.

15.2 SEPARACIÓN DINÁMICA DE ASPECTOS

Como ya se ha visto anteriormente, la programación orientada a aspectos se basa en la inserción de un determinado código en puntos específicos marcados de un programa (antes o después de la ejecución de un método, etc.). El empleo de reflexión estructural permitiría obtener una funcionalidad similar, ya que podremos introducir llamadas a métodos nuevos en el punto que deseemos y también retirarlos cuando sea necesario, mediante el análisis del código de un método potencialmente instrumentable de esta forma y su posterior modificación. Esto permitiría pues ejecutar el código que deseemos sin necesidad de predeterminar los puntos en los que queremos introducir dicho código. Este método es más potente ya que, al no ser necesario marcar los puntos de inserción de antemano, podremos insertar el código en cualquier lugar y de esta forma lograr una mayor flexibilidad.

Por tanto, podemos concluir que la reflexión estructural empleada de esta forma puede tener las ventajas que posee la programación orientada a aspectos dinámica, logrando además una mayor flexibilidad. Nuestro sistema podría ofrecer estas funcionalidades de forma más eficiente, además de poder aplicar estas ventajas a cualquier lenguaje de la plataforma, aunque inicialmente no estuviese pensado para soportar estas características. Un posible trabajo futuro consistiría en comparar las diferencias de rendimiento entre un lenguaje que implemente *DAOP* (*Dynamic Aspect Oriented Programming*) frente a este sistema. Existen iniciativas en este sentido, a través del proyecto futuro que veremos usando el sistema *Phoenix* de *Microsoft*.

15.3 DEBUGGING Y PROFILING EN TIEMPO DE EJECUCIÓN

Se podrían hacer labores de *profiling* o de depuración de cualquier programa que deseemos mediante una variante de la técnica descrita en el punto anterior, introduciendo en puntos concretos métodos cuya finalidad sea precisamente comprobar condiciones concretas del programa, valores de variables y, en definitiva, cualquier condición que deseemos observar, con la posibilidad de eliminarlos en un momento dado para que el programa funcione con su máxima eficiencia, deshaciéndonos de estos elementos introducidos automáticamente cuando dejen de ser necesarios. Por tanto, la principal ventaja de esta forma de instrumentar un programa es que podemos eliminar la instrumentación sin necesidad de parar el programa y volver a compilarlo, además de contar con una mayor flexibilidad a la hora de aplicar estas técnicas.

15.4 IMPLEMENTACIÓN DEL MODELO RDM

RDM (*Rapid Domain Modelling*) [Izquierdo02] es un *framework* desarrollado en la Universidad de Oviedo para crear una arquitectura que permita modelar los distintos elementos del dominio de una aplicación. Este *framework* se basa en que, dado que la forma de construir las entidades de una aplicación, así como sus relaciones y operaciones, es consecuencia de un conjunto de decisiones que pueden estar sometidas a condiciones muy variables (al menos en los primeros ciclos del desarrollo), es necesaria la creación de una técnica de implementación y diseño ágil que permita seguir el ritmo de cambio constante que se exige al desarrollador.

Podemos decir de forma abreviada que este *framework* crea una serie de entidades y conceptos relacionados para modelar aplicaciones, cuya máxima de funcionamiento es lograr un nivel de flexibilidad lo más elevado posible, permitiendo modificar entidades y relaciones entre ellas de manera dinámica de forma sencilla, y adaptándose de esta forma a los cambios en el *software* de manera adecuada. La flexibilidad que posee este modelo es similar a la que poseen lenguajes dinámicos dotados de reflexión estructural que hemos descrito en esta tesis (añadir miembros a entidades, cambiar relaciones entre ellas, etc.), por lo que creemos que este *framework* es completamente implementable sobre nuestra plataforma. Además, muchos de los elementos que forman parte del mismo son fácilmente traducibles a conceptos propios de lenguajes dinámicos como los mencionados.

Este *framework* define también otro conjunto de conceptos (monitores, reacciones y reconstructores), apoyados sobre un sistema de eventos, que se encargan de controlar ciertos aspectos de la aplicación creada (como la validación de las reglas de negocio, recuperación del sistema en caso de que ocurra alguna operación errónea, etc.) de manera completamente separada de las entidades de la aplicación. Estos conceptos también son plenamente implementables sobre nuestro sistema, ya que se basan en el mencionado sistema de eventos y en manipular dinámicamente la estructura y comportamiento de las entidades. Pueden consultarse en detalle todos los aspectos de este *framework* en [Izquierdo02]

Como hemos mencionado anteriormente, todos los elementos que constituyen *RDM* son implementables sobre nuestra máquina virtual extendida. No obstante, hacerlo aporta algunas ventajas como son:

- Una posible mayor eficiencia del sistema en general, al estar implementado sobre una base que soporta operaciones dinámicas de forma nativa y les da una mayor eficiencia. Dado que el todo el sistema está basado en estas operaciones dinámicas, es razonable pensar que la eficiencia será mayor al no tener que emularlas sobre un lenguaje estático "tradicional".
- Contar con todas las ventajas que nos proporciona el uso de la máquina virtual *SSCLI*, a destacar sobre todo la interoperabilidad de lenguajes que permitiría a *RDM* usar cualquier clase desarrollada en un lenguaje de la máquina.
- Permitir a las aplicaciones emplear el modelo *RDM* para el desarrollo de todo o parte de su código, en función de las necesidades de cada módulo de la misma.

15.5 IMPLEMENTACIÓN DE FRAMEWORKS DE APLICACIONES

Actualmente existen diversos *frameworks* de aplicaciones como *Ruby on Rails* [Thomas05], *Zope* [Zope06] o *Django* [Django06] cuya principal ventaja es que, mediante el uso de metaprogramación (introspección y reflexión estructural), permiten la creación de aplicaciones de una forma más rápida, aumentando potencialmente, sin perder capacidades en otros aspectos, la productividad de las empresas que usen este tipo de sistemas. Sin entrar en detalles excesivos, podemos afirmas que estos sistemas emplean la reflexión para automatizar la creación de código de distintas partes de la aplicación que se está creando, librando al programador de hacer tareas repetitivas múltiples veces (principio *DRY: Don't Repeat Yourself* [Venners03] y permitiendo así una creación más rápida del sistema.

Debido al éxito de este tipo de *frameworks*, lenguajes de relativamente nueva creación están desarrollando iniciativas similares (ver *Grails* [Grails06] para el lenguaje *Groovy*), por lo que puede ser interesante intentar hacer una iniciativa similar sobre el sistema extendido diseñado.

Por tanto, dado que el sistema extendido ofrece un soporte de reflexión similar al que los lenguajes con los que se han construido estos *frameworks* poseen, no habría teóricamente ningún inconveniente en crear un sistema de similares características sobre el sistema extendido, que además gozaría de las ventajas que suponen tener un mayor rendimiento y todas las características que ofrece la máquina virtual sobre la que se ha creado el mismo.

15.6 APLICACIÓN DE ЯROTOR CON MDA

En esta sección vamos a estudiar cómo el sistema ampliado puede tener aplicación en el campo de Desarrollo de Software Dirigido por Modelos (*MDD*) y más concretamente *MDA* (*Model-Driven Architectures* o arquitecturas dirigidas por modelos). Veremos también cómo puede usarse para obtener ventajas de rendimiento en conjunto con herramientas que sigan la filosofía *MDA*. Para ello, haremos primero una descripción muy breve de lo que es *MDA* y sus herramientas, sin profundizar en estos aspectos. Puede ampliarse el conocimiento de los diferentes conceptos relacionados con *MDA* en la bibliografía que se proporciona como referencia.

La propuesta *Model Driven Architecture* (*MDA*) del *Object Management Group* (*OMG*) [OMG07] está tomando cada vez mas fuerza en el desarrollo de software actual [Pelechano04]. Esta propuesta se ha creado con la intención de tratar de mejorar la calidad del *software* desarrollado, facilitando su portabilidad, interoperabilidad y reusabilidad. *MDA* promueve el uso intensivo de modelos en el proceso de desarrollo de *software*. Siguiendo esta aproximación, los desarrolladores construirán modelos de los sistemas *software* utilizando primitivas de alto nivel de abstracción, que posteriormente serán transformados sucesivamente hasta obtener el código del sistema *software* final.

15.6.1 Naturaleza de MDA

La Guía de *MDA* [Mukerji01] especifica que "*MDA is an approach to using models in software development*". No obstante, conviene hacer una distinción entre éste acrónimo y *MDD*. Ciertos trabajos [Pelechano04] establecen la distinción entre *MDA* y *MDD* (*Model Driven Development*) definiendo este último como una aproximación al desarrollo de software basada en el modelado del sistema, y su generación posterior a partir de estos modelos creados del mismo. No obstante, al ser únicamente una aproximación, solo proporciona una estrategia general a seguir en el desarrollo de *software*, pero no define ni técnicas a utilizar, ni fases del proceso, ni ningún tipo de guía metodológica.

Aquí es donde entra en juego precisamente *Model Driven Architecture* (*MDA*), que es un estándar de *OMG* que defiende el *MDD* y agrupa varios lenguajes que pueden usarse para seguir esta aproximación. Según esto, *MDA* posee pues el valor añadido de proporcionar lenguajes con los que definir métodos que sigan *MDD*. Sin embargo, *MDA* no es por sí mismo un método que define técnicas, etapas, artefactos, etc., solamente proporciona la infraestructura tecnológica y conceptual con la que construir estos métodos que siguen *MDD*.

Antes de continuar, es preciso definir muy brevemente la terminología más importante que reside en la propuesta *MDA*, para enunciar mejor la relación con nuestro sistema. Según el *OMG* [OMGMDA07], *MDA* es una nueva forma de desarrollar aplicaciones y escribir especificaciones, basada la creación de un modelo de la aplicación o funcionalidad que se pretende construir que sea independiente de la plataforma (*PIM* o *Platform Independent Model*). Una especificación de *MDA*, además del *PIM* mencionado, también está formada por uno o más *PSM* (*Platform Specific Model*), que son modelos específicos del sistema destinados a una plataforma concreta (existiendo uno por cada plataforma). Además, también existirán una serie de implementaciones completas de estos modelos, de las que también habrá una por cada plataforma donde los desarrolladores deseen implementar el sistema. La siguiente figura muestra el esquema de los distintos componentes de un *MDA*. Este esquema es muy simple y solo muestra los principales componentes y sus transformaciones de forma básica. Existen, como se mencionará posteriormente, otras transformaciones posibles entre ellos.

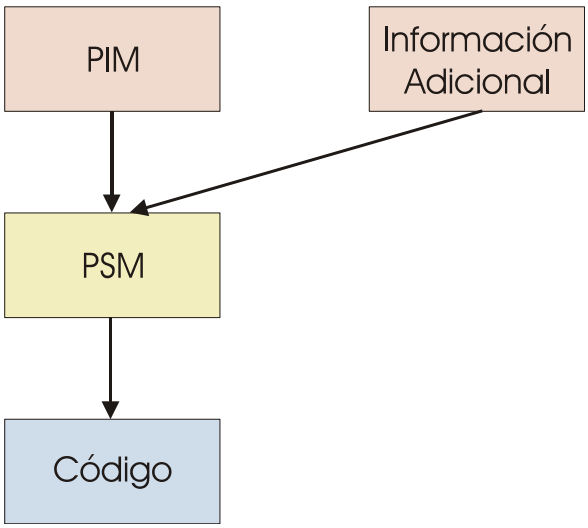

Figura 15.1: Esquema de componentes de *MDA*

El desarrollo basado en *MDA* prioriza la funcionalidad y el comportamiento de la

aplicación a construir sin que en ello intervengan otros aspectos, como la tecnología a usar o las particularidades de la plataforma donde va a ser implementada. De esta manera, *MDA* trata de separar completamente los detalles de implementación de las reglas de negocio, de forma que la aparición de una nueva tecnología no requiera repetir la especificación de toda la funcionalidad y comportamiento de la aplicación, que solo será necesario modelar una vez. A partir de esta funcionalidad se construirá el *PIM*, que se convertirá a un *PSM* específico de una plataforma que soporte *MDA*, usando para ello herramientas que permitan hacer esta transformación (y que finalmente permitirán la generación de código plenamente funcional en dichas plataformas).

En definitiva, para que *MDA* pueda ser utilizado directamente para desarrollar software, es necesario que se construyan métodos precisos que proporcionen a los equipos de desarrollo pautas y técnicas que éstos puedan seguir y utilizar. Por tanto, para que *MDA* resulte realmente útil para el desarrollo de *software* real, es necesario definir concretamente elementos como los pasos a dar al desarrollar software usando esta propuesta, los métodos que incluye la misma o profundizar en las transformaciones entre modelos, de modo que los desarrolladores tengan claros los requisitos que deben satisfacer, para poder especificar transformaciones automáticas y lograr así una alta automatización (idealmente completa) de la generación de código a partir de los datos del modelo del sistema creado previamente.

15.6.2 Aspectos de las Herramientas que Soportan MDA

Actualmente existen diversas herramientas que proporcionan soporte para los conceptos y la aplicación de *MDA*. La característica más importante de las mismas es el soporte que poseen para efectuar transformaciones entre modelos, que permitirá obtener un modelo "destino" a partir de un modelo "origen". No obstante, aunque para permitir transformaciones entre modelos complejas es necesario que éstas estén especificadas de forma clara y sin ambigüedades, aun no existe un estándar para ello en la actualidad, aunque hay uno en desarrollo denominado *QVT* [OMGQVT07]. La especificación de una transformación define como se comportará una herramienta con soporte para *MDA* cuando la haga. En general, pueden distinguirse dos tipos de transformaciones que realizan estas herramientas, y que permiten clasificarlas en dos grupos:

- **Herramientas de transformación modelo-modelo**: Que, siguiendo la terminología *MDA*, podrían ser *PIM-PIM* (para transformar de un modelo independiente de la plataforma a otro similar pero más especializado, mejorado o filtrado de alguna forma), *PIM-PSM* (para pasar el modelo independiente a uno específico de una plataforma), *PSM-PSM* (al igual que *PIM-PIM*, permite refinar el modelo *PSM*) y *PSM-PIM* (que permitiría crear modelos más abstractos a partir de modelos *PSM* existentes). Ejemplos de estas herramientas son *OptimalJ* [OptimalJ07] o *ArcStyler* [ArcStyler07], entre otras.
- **Herramientas de transformación modelo-código**: Estas herramientas se encargan de transformar un modelo en código que siga una determinada gramática de un lenguaje concreto. En algunas herramientas la transformación de *PIM* a *PSM* consiste precisamente en generar el código del programa directamente.

Actualmente no existe ninguna herramienta que soporte todos los requisitos de *MDA*, principalmente debido a que no existe todavía un estándar para hacer las transformaciones entre modelos y que en la actualidad no hay demanda para ciertas

características poseídas por los *MDA* [Pelechano04]. Por tanto, muchos fabricantes de herramientas implementan las transformaciones, reglas y demás elementos necesarios para dar soporte a *MDA* de una forma propietaria, dada la carencia de estándar.

Como ejemplo de un sistema concreto mencionaremos la herramienta *OptimalJ* [Igamberdiev04], que permite desarrollar las aplicaciones separando las reglas de negocio del resto del código de la aplicación. Las reglas de negocio están definidas en los modelos, a un nivel alto de abstracción que permita realizar esta separación. *OptimalJ* traduce estas reglas de negocio a una aplicación *J2EE* en funcionamiento, y propaga los cambios que se realizan en las mismas a los puntos adecuados de la aplicación, asegurándose de que se cumplen todas las reglas especificadas en las diferentes capas de la misma.

Las reglas de negocio en este sistema están mantenidas y monitorizadas de forma independiente del resto del sistema, usando un repositorio de reglas y accediendo a ellas a través de un servidor de reglas de negocio, centralizando su manejo en un único punto y permitiendo así su reutilización más sencilla. Este servidor permite realizar un acceso dinámico a estas reglas, de manera que cualquier desarrollador, o usuario con permisos necesarios para ello, pueda cambiar directamente cualquier regla de negocio y permitir que esos cambios se vean reflejados inmediatamente en las operaciones realizadas por la aplicación. Para ello, cada vez que una aplicación necesita hacer uso de una regla, accederá a ella a través del servidor de reglas de negocio, independientemente de la capa en la que se encuentre el componente que necesite usarla. De esta forma, el sistema *OptimalJ* permite que los sistemas que se construyan con él tengan un alto nivel de adaptabilidad, ya que en todo momento podrá tener actualizadas las reglas de negocio que debe seguir.

Otras herramientas interesantes que siguen la propuesta *MDA* son *AndroMDA* [AndroMDA07], de libre distribución, y *BOA* [Padrón04], que es una herramienta surgida a partir de y aplicada en un proyecto real y que sigue la filosofía *MDA*, pero presentando algunas diferencias con las estrategias planteadas por el *OMG*.

15.6.3 Relación de MDA y ЯRotor

La razón por la que la propuesta *MDA* y el sistema desarrollado en esta tesis pueden estar relacionados, de forma que se pueda lograr una mejora para algún aspecto de *MDA* usando nuestro sistema, reside precisamente en la propia forma de funcionamiento de *MDA*. Anteriormente hemos visto cómo la herramienta *OptimalJ* ofrece un soporte para que un sistema dado pueda reaccionar rápidamente a cambios en sus reglas de negocio, de manera que se aumente su flexibilidad. Este tipo de facilidades creemos que pueden mejorarse, y describiremos en esta sección una posible vía para ello.

Si la implementación final de un sistema es el resultado de hacer una serie de transformaciones de su modelo *PIM*, cualquier cambio existente en este modelo (motivado por la introducción de nuevos requisitos, características, restricciones, reglas de negocio, etc.) necesitará la generación de nuevos modelos que transformen el *PIM* modificado, siguiendo el mismo proceso que se realizó para transformar el *PIM* original y obtener el código de la implementación [Binstock07]. Esto es necesario para que finalmente se pueda obtener de nuevo el código funcional de la aplicación, pero con todos los cambios al modelo *PIM* que se han realizado integrados en el mismo.

Por tanto, al ser el código de la aplicación un proceso resultante de la transformación de un modelo, cualquier cambio en ese modelo necesitaría la repetición del proceso de transformación del mismo en código para reflejar este cambio. La mayoría de cambios introducidos en un momento dado en el modelo no podrán ser incorporados

directamente sobre el código de la aplicación final si el lenguaje de destino es estático. Estos lenguajes, como hemos visto en esta tesis, no ofrecen facilidades potentes para la manipulación del contenido de sus instancias, clases o relaciones entre las mismas de manera dinámica, por lo que será necesario parar y recompilar la aplicación obtenida del proceso de transformación para poder incorporarlos.

Si bien, como ya se ha mencionado con *OptimalJ*, existen aproximaciones que sí que gozan de un determinado grado de flexibilidad y adaptabilidad, esta flexibilidad está limitada en cuanto a las entidades que afecta o lo que se puede hacer con ella (en este caso se limita a reglas de negocio), y la aplicación en sí podría tener que interactuar explícitamente con el *framework* de la herramienta que soporte *MDA* para contemplar estos cambios (como vimos también en *OptimalJ* con su servidor de reglas de negocio).

No obstante, un lenguaje dinámico, que goce capacidades de reflexión estructural, podría incorporar un número de cambios más elevado a la aplicación en tiempo de ejecución, sin ser necesario que se detenga la ejecución del sistema. Una herramienta *MDA* que genere código para un lenguaje dinámico no necesitaría parar la ejecución del programa y hacer todo el proceso de transformación de nuevo cuando ocurra un cambio en el modelo. Bastaría con que se registrase de alguna forma qué elementos han sido modificados respecto al modelo original y alterar la estructura de la aplicación de forma acorde, creando nuevas entidades, métodos y cualquier entidad que se juzgase necesaria para contemplar los cambios realizados al modelo. Con ello se logra por tanto una doble ventaja:

- **Un proceso más eficiente**: Ya que no es necesario reconstruir todo el sistema (y las transformaciones que llevan a él) cada vez que ocurran cambios al modelo, sólo incorporar al mismo exclusivamente los cambios realizados.
- **Adaptación dinámica**: El sistema no tendría que parar necesariamente su funcionamiento, y podría adaptarse dinámicamente a cualquier cambio que ocurra en su modelo de forma, mejorando así su flexibilidad y adaptabilidad al ser capaz de evolucionar de manera mucho más rápida ante las nuevas condiciones introducidas.
- **Independencia del Lenguaje**: Anteriormente hemos visto como nuestro sistema ha definido la forma en la que lenguajes dinámicos y estáticos que se ejecuten sobre el mismo puedan interoperar. Esto, unido a la interoperabilidad poseída por la plataforma, hace que sea posible la generación de código para cualquiera de estos lenguajes, y que dicho código pueda colaborar para la construcción de una aplicación completa. Esto hace que se pueda generar código dinámico o estático en función de que tipo de lenguaje sea más adecuado para un determinado módulo de la aplicación, adaptándose a las necesidades concretas de cada parte de la misma y dotando de un mayor nivel de flexibilidad solo a aquellas partes en las que este previsto que sea necesaria. Un *MDA* que genere código para nuestra plataforma podría pues generar código dinámico o estático para una misma aplicación allí donde lo desee, y por tanto podría generar código dinámico en aquellas partes donde se considerase oportuno (para aprovechar su flexibilidad), así como código estático en aquellas partes donde sus ventajas (como obtener un mayor rendimiento) sean apreciables. El *MDA* sería, efectivamente, independiente del lenguaje.

Por tanto, un lenguaje dinámico parece plantear diversas ventajas teóricas a la hora de trabajar con arquitecturas dirigidas por modelos, ya que permiten contemplar mejor la adaptabilidad que permite la propuesta *MDA* por el hecho de tener un modelo *PIM*, que nos da la posibilidad de contemplar el sistema sin la presencia de su implementación y que se pueda actuar de esta forma directamente sobre el modelo del sistema para hacer cambios.

No profundizaremos más en detalle acerca del uso de lenguajes dinámicos y *MDA* y que ventajas o inconvenientes plantean uno sobre el otro [Eckel04], ya que está fuera del ámbito de ésta tesis. A modo de resumen, podemos afirmar que, al dar nuestro sistema un soporte más eficiente a los lenguajes dinámicos, una herramienta *MDA* que generase código para un lenguaje dinámico que se ejecutase sobre el mismo podría generar código a partir de un *PIM* que tuviese un rendimiento mejor. Además, como hemos visto, las primitivas de reflexión añadidas a nuestro sistema hacen que se soporte la incorporación dinámica de los cambios introducidos al modelo en un momento dado (sin importar el lenguaje en el que este creada la aplicación, puesto que las primitivas son comunes a todos ellos), y su interoperabilidad puede también mejorar el rendimiento de la aplicación final a la vez que posee un elevado nivel de flexibilidad.

16 CONCLUSIONES

16.1 CONSECUCIÓN DE LOS REQUISITOS

A continuación describiremos cómo el sistema construido se ajusta a los requisitos establecidos al principio del documento.

16.1.1 Requisitos Arquitectónicos

- **Máquina virtual**: La máquina virtual escogida permite la modificación del código de cualquier módulo de la misma e incorporar cualquier cambio que se le haga de forma sencilla. Además, su licencia de uso permite hacer casi cualquier tipo de modificación (de hecho, existen múltiples proyectos de modificaciones de distinta índole, financiados por la propia *Microsoft Research*) y documentación suficiente para las modificaciones a los distintos módulos que sean necesarias. Por último, la máquina es estable, no está sometida a evoluciones frecuentes y cuenta con un compilador *JIT*, que cubre nuestras necesidades de optimización en tiempo de compilación.
- **Independencia del Lenguaje de Programación**: La máquina escogida soporta varios lenguajes y permite añadir lenguajes nuevos que compilen código para la misma, al estar basada y seguir fielmente el estándar *CLI*.
- **Independencia del Problema**: La máquina escogida es de propósito general.
- **Soporte para Capacidades Reflectivas**: La máquina seleccionada cuenta con capacidades de reflexión. La introspección y programación generativa soportadas por dicha máquina se han juzgado suficientes para cumplir con el requisito expresado en este sentido, al ser una buena base a partir de la cual desarrollar la extensión de las capacidades reflectivas.
- **Independencia de la Plataforma y Portabilidad**: La máquina seleccionada y sobre la que se han hecho todas las modificaciones está pensada para facilitar su portabilidad a otras plataformas (de hecho, ya se ha mencionado que originalmente soporta varias). A lo largo del trabajo realizado para incorporar las modificaciones al sistema, se ha tenido especial cuidado en no interferir con los distintos mecanismos para facilitar la portabilidad del sistema que tiene integrados. Por otra parte, el hecho de que todo lenguaje de la máquina (ya existente o incorporado posteriormente) comparta el mismo código *CIL* estandarizado garantiza que cualquier programa se ejecutará en cualquier implementación de la máquina virtual sobre un *hardware* concreto, cumpliéndose así este requisito.
- **Proyección del Sistema para el Desarrollo de Aplicaciones Reales**: Si bien el sistema seleccionado no se usa como base para el desarrollo de ninguna aplicación real notable hasta el momento, la máquina virtual *CLR*, con la que guarda una

gran similitud arquitectónica, es hoy en día una de las máquinas que se usa cada vez más como soporte de un número mayor de aplicaciones de distinto tipo, como parte del *framework .NET*. Por ello, y dado que el código de esta última máquina no es abierto, se ha juzgado adecuado el uso de *SSCLI* para poder así comprobar la idea subyacente en esta tesis con un sistema muy similar a uno comercial y permitir, en un futuro, estudiar su incorporación en el mismo.

Vemos de esta forma cómo la máquina virtual seleccionada es la que mejor cumple con todos los requisitos fijados al comienzo de esta tesis, considerándose pues que se ha escogido el mejor sistema de partida disponible y cumpliéndose así este requisito.

16.1.2 Requisitos de Compatibilidad del Sistema

- **Conservar las Características del Sistema**: El sistema ha sido modificado ampliando y no sustituyendo ninguno de sus módulos. Se ha puesto especial cuidado en usar todas las herramientas que el sistema ya poseía para desarrollar sus nuevas funciones, y en todo momento se tratado de conservar el código existente en el caso de que no se usasen las capacidades de reflexión. Una de las partes más delicadas al respecto, la generación de código nativo con nueva semántica a partir del *CIL*, se ha hecho usando las macros de generación de código ya existentes, por lo que el sistema no se verá afectado al respecto (portabilidad). La no modificación de los lenguajes que maneja la máquina también contribuye a mantener las características del sistema original. Por tanto, dada la implementación hecha y que las numerosas pruebas realizadas, incluyendo los *benchmarks* y programas de prueba vistos, no han relevado ningún problema al respecto, puede afirmarse que las características del sistema (portabilidad, etc.) se han conservado.
- **Compatibilidad hacia Atrás**: Además de lo dicho en el punto anterior, el conjunto de programas de prueba con el que se ha trabajado, y el especial cuidado que se ha puesto en este aspecto, permite afirmar que el sistema extendido sigue pudiendo ejecutar los mismos programas que el original sin que la ejecución de los mismos devuelva valores diferentes o errores.
- **Ampliación del Modelo Computacional para Soportar Lenguajes Dinámicos**: A lo largo de esta tesis hemos estudiado cómo el modelo computacional del sistema se ha ampliado para soportar las primitivas de reflexión sobre las que se apoyan los lenguajes dinámicos. Dado que hemos conseguido integrar dichas primitivas en el sistema satisfactoriamente, podemos afirmar que el nuevo modelo computacional desarrollado permite soportar lenguajes dinámicos de forma nativa en la máquina ampliada.
- **Interoperabilidad Ampliada**: En un capítulo anterior se ha mostrado cómo el nuevo modelo computacional permite la ejecución de lenguajes dinámicos y estáticos de forma colaborativa, sin perjuicio de ninguno de los dos tipos. Ahora no sólo los lenguajes estáticos de la máquina podrán cooperar, sino que también lo podrán hacer los dinámicos y se ha contemplado, de la forma vista, el establecer colaboraciones entre lenguajes de diferente tipo, gracias al modelo desarrollado.

Por tanto, estamos en disposición de afirmar que el sistema modificado mantiene una compatibilidad total con el código heredado y que permite la "convivencia" de lenguajes que usen ambos modelos computacionales, satisfaciéndose este requisito.

16.1.3 Requisitos Reflectivos

- **Introspección**: El sistema escogido como base (*SSCLI*) ya contaba con esta característica, que evidentemente se sigue conservando.
- **Modificación Dinámica de la Estructura del Sistema**: El modelo computacional diseñado está preparado para admitir las primitivas de reflexión estructural, lo que asegura la capacidad de modificar cualquier tipo de miembro (atributo o método) de una clase o una instancia, y permite dar un soporte adecuado para los lenguajes dinámicos que se deseen implementar sobre la máquina extendida. No obstante, en el prototipo del sistema que se presenta en esta tesis falta por implementar por el momento la primitiva de cambio de clase de una instancia, algo que se hará en un futuro mediante otro proyecto que ya está en marcha. Por tanto, las nuevas funcionalidades que han sido integradas en el motor de ejecución dotan al sistema de un nivel de flexibilidad superior al existente y suficiente para considerar que este requisito ha sido cumplido.

Podemos afirmar entonces que el sistema extendido soporta un nivel de reflexión equiparable al que ofrecen los lenguajes dinámicos más usados (Ej.: *Python*, *Ruby*) y que este requisito se ha visto satisfecho adecuadamente.

16.1.4 Requisitos de Eficiencia

A partir de los resultados mostrados en el capítulo de evaluación del sistema, podemos afirmar lo siguiente:

- **Rendimiento Frente a Sistemas de Características Similares**: El rendimiento del sistema extendido se ha mostrado superior al de sistemas de características similares (seleccionando para ello el lenguaje que ofrecía un mejor rendimiento en general dentro de todos aquéllos que podrían compararse con nuestro sistema), usando código que empleaba capacidades de reflexión, con un coste de memoria justificable.
- **Penalización de Rendimiento Respecto al Sistema de Partida**: El rendimiento del sistema original se ha visto disminuido por las modificaciones realizadas, pero dicha disminución se ha considerado asumible dadas las ventajas obtenidas, y se ha constatado que será posible reducir aún más el coste de la implementación de las nuevas capacidades en el sistema, mediante la incorporación o perfeccionamiento de nuevas optimizaciones.
- **Eficiencia de las Optimizaciones Realizadas**: En todo caso se ha comprobado que cualquier ganancia de rendimiento que el sistema ofrece se ha hecho con un coste en memoria razonable, tanto con operaciones reflectivas como con operaciones no reflectivas.

Por tanto, podemos considerar que los requisitos de eficiencia, a tenor de los resultados obtenidos, se han cumplido satisfactoriamente, validando la idea subyacente en esta tesis.

16.2 CONCLUSIONES FINALES

A continuación se expondrán las conclusiones que se han podido extraer tanto del desarrollo del sistema como de los resultados obtenidos. Las máquinas abstractas se han usado ampliamente para el diseño y construcción de lenguajes de programación debido a que ofrecen un buen número de ventajas, tal y como hemos visto en esta tesis. Uno de los principales inconvenientes de estos sistemas, su bajo rendimiento, está tratando de solventarse mediante la investigación, desarrollo e implementación de técnicas como la compilación adaptativa (*Hotspot*) *Just In Time*, que incrementan el rendimiento final de este tipo de sistemas de forma significativa y que hacen que hoy en día sean muy usados comercialmente.

Actualmente, los lenguajes dinámicos son un medio de desarrollo cuya popularidad y ámbito de uso está aumentando en diferentes escenarios de la ingeniería del *software*, sobre todo aquéllos que requieren unas necesidades altas de flexibilidad y adaptabilidad. Estos lenguajes suelen hacer uso de máquinas abstractas, compilándose a un código intermedio propio de la misma para su ejecución. Las especiales características de flexibilidad de los lenguajes dinámicos hacen que su semántica sea también más compleja, lo que normalmente deriva en que su implementación deba hacerse mediante un intérprete, que permite desarrollar un soporte adecuado para su flexibilidad de una manera más sencilla. Esta flexibilidad también convierte la generación de código nativo usando *JIT* en una tarea más difícil. Por todo ello, el rendimiento de los lenguajes dinámicos generalmente es significativamente peor que el de los estáticos, siendo éste su principal inconveniente.

En un intento de mejorar su rendimiento, existen diferentes implementaciones de lenguajes dinámicos (como *Python*) para plataformas como *.NET* o *Java*, que tratan de aprovechar la compilación *JIT* de sus máquinas virtuales para obtener un mejor rendimiento. Dado que estas máquinas poseen un modelo computacional orientado a objetos no pensado para operaciones reflectivas, se debe usar una capa de abstracción adicional que simule las características de reflexión sobre ellas. Esta capa adicional hace que para la ejecución de una capacidad reflectiva cualquiera se necesite código adicional que causa un impacto significativo en el rendimiento. Mientras que estos sistemas que emulan las capacidades reflectivas ofrecen un rendimiento mejor en programas no reflectivos (aunque para estos programas el mejor medio de desarrollo es precisamente un lenguaje no reflectivo), el uso de capacidades reflectivas, tal y como se ha visto en las mediciones realizadas, tiene un rendimiento pobre. Hemos evaluado sistemas de este tipo (como *Jython* e *IronPython*) además de otras implementaciones no basadas en esta clase de máquinas virtuales, sino implementadas directamente en *C* (como *CPython* [CPython06] y *ActivePython* [ActivePython06]), obteniendo mucho mejor rendimiento en las segundas que en las primeras cuando se usan sus características reflectivas (*Jython* es casi 29 veces e *IronPython* es 428 veces más lento que *CPython*).

No obstante, la conclusión a la que se llega en esta tesis es que realmente la combinación de máquina abstracta y compilador *JIT* sí constituye un medio para aumentar el rendimiento de los lenguajes dinámicos. Para ello, en lugar de generar código adicional para simular las capacidades dinámicas sobre una máquina estática, como hacen los sistemas mencionados, hemos ampliado la máquina virtual *SSCLI* para soportar nativamente las primitivas reflectivas de los lenguajes dinámicos. El modelo de orientación a objetos basado en clases de esta máquina virtual se ha extendido con la semántica de un modelo de orientación a objetos basado en prototipos, donde las clases representan *trait objects*. Para poder acceder a las nuevas primitivas reflectivas se han desarrollado una serie de servicios que las representan, integrados dentro del motor de la máquina y accesibles tanto desde la librería estándar de *SSCLI* (*BCL*) como desde el propio código intermedio *CIL* de los programas generados, de manera que el uso de las mismas pueda ser lo más transparente posible y pueda quedar adecuadamente integrado

en la máquina de partida manteniendo la compatibilidad con el código heredado. Todas estas primitivas implementan la semántica del modelo de orientación a objetos basado en prototipos, y han sido creadas con la intención de que no interfieran con el modelo ya existente, "conviviendo" pues con el modelo basado en clases "tradicional" que posee el sistema base, *CLI*. Dependiendo del lenguaje de alto nivel que vaya ser compilado, el compilador usará el modelo basado en clases o el modelo basado en prototipos, usando al mismo tiempo un sistema de tipos estático o dinámico según sus necesidades.

El rendimiento obtenido con nuestro sistema *SSCLI* extendido (*ЯRotor*) demuestra que es más rápido con código reflectivo (unas 3 veces más rápido que *CPython* [CPython06]), con un consumo de memoria adecuado. En cuanto a la ejecución de código no reflectivo, la optimización agresiva para la generación de código no reflectivo del compilador *JIT* de *SSCLI* hace que nuestro sistema extendido supere ampliamente en rendimiento a las implementaciones de *Python* con las que lo hemos comparado, ocupando sólo un 28% más de recursos de memoria.

Por último, el coste impuesto al sistema base por las modificaciones que se le han realizado, cuando se ejecutan programas reales que no usan reflexión, es cercano al 100% usando un 7% de memoria adicional. De estos resultados se extrae la conclusión de que el coste en rendimiento derivado de la modificación de *SSCLI* se debe a su diseño original como plataforma estática. La máquina virtual original ha sido implementada haciendo optimizaciones muy agresivas de cara a lograr el mejor rendimiento posible para lenguajes estáticos. De haber sido diseñada desde un principio con ambos modelos en mente, el rendimiento medio de los *benchmarks* realizados (estáticos o dinámicos) probablemente hubiera sido significativamente mejor. En el futuro se trabajará en mejorar la optimización del sistema y completar el soporte para lenguajes dinámicos implementando más servicios a partir de las primitivas y el modelo desarrollado en esta tesis.

Por tanto, la conclusión final de este trabajo es que el uso de una plataforma basada en una máquina abstracta, diseñada para soportar características reflectivas nativamente (no emuladas), junto con técnicas de optimización dinámica (como *JIT*), ofrece mejoras de rendimiento importantes para los lenguajes dinámicos, pudiéndose crear una máquina con un soporte adecuado adicional para los lenguajes con sistemas de tipos estáticos y un sistema que soporte eficientemente ambos tipos de lenguajes, la interoperabilidad entre ellos, y con las ventajas inherentes a las máquinas virtuales.

17 TRABAJO FUTURO

Además de todo lo mencionado en el capítulo de "Aplicaciones del sistema desarrollado", que muestra posibles vías de desarrollo de nuevas aplicaciones a partir de lo que se ha desarrollado en esta tesis, se describirán ahora otros posibles desarrollos orientados a perfeccionar y ampliar la arquitectura desarrollada, sus componentes y las partes de la misma, más que con sus posibles aplicaciones prácticas. A continuación se muestra una selección de posibles propuestas de ampliación, descritas brevemente para mostrar cómo se podrían hacer y qué cosas se deberían tener en cuenta.

17.1 MIGRAR LA IMPLEMENTACIÓN A OTRAS PLATAFORMAS

En el tema que se ocupó en su momento de estudiar los posibles sistemas candidatos para hacer la modificación planteada en esta tesis, se vio que existían otras plataformas cuya arquitectura y ámbito de aplicación era similar al escogido finalmente. Por ello, una vez demostrado que es posible modificar un sistema de esta clase para poder incorporar las modificaciones planteadas, y que estas modificaciones devuelven unos resultados satisfactorios, cabe la posibilidad de aplicar los mismos cambios a los otros sistemas cuyo código es accesible, *Mono* y el proyecto *DotGNU*. Por ello, siempre que se llegue a una versión que se pueda considerar estable de cualquiera de esas plataformas (que no esté sometida a evoluciones constantes), se puede plantear la modificación de las mismas siguiendo la filosofía de diseño planteada para *SSCLI*, dotando a dichas plataformas de las mismas capacidades y funcionalidades. En caso de lograrlo, podríamos tratar de constatar los siguientes aspectos:

- Que la arquitectura y el diseño de las modificaciones realizadas, el modelo computacional y los cambios del motor son adecuados al poder aplicarse a otras máquinas virtuales. Siempre y cuando se puedan integrar correctamente en estas otras plataformas las operaciones reflectivas, aun teniendo que hacer cambios en la forma de implementar las operaciones, por las más que probables diferencias existentes en la forma de diseñar las estructuras de datos que soporten todas las funcionalidades que ofrecen, se ratificará que el modelo diseñado es aplicable de forma general a máquinas abstractas.
- Que el principal objetivo pretendido en esta tesis (dotar de más eficiencia a los lenguajes dinámicos) se vea corroborado, al conseguir que sistemas basados en máquinas virtuales de implementaciones muy distintas obtengan resultados similares (o mejores) a plataformas dinámicas existentes con las que se comparen.

Por otra parte, también se puede plantear esta modificación a otras máquinas virtuales que no estén relacionadas con la plataforma *.NET* (*Java* o *Smalltalk*).

17.2 CONSTRUIR UNA NUEVA MÁQUINA

Como ya se ha dicho a la hora de describir la implementación del prototipo, a medida que se construían dentro del *SSCLI* las modificaciones planteadas, gran parte del trabajo desarrollado se vio afectado por la especial arquitectura de la máquina, pensada para la máxima eficiencia de lenguajes estáticos. Hemos tenido que tomar decisiones de implementación que dificultan la implementación de algunas de las operaciones dinámicas que se tratan de construir, lo que puede tener un coste en tiempo y recursos no desdeñable.

Es por ello que, con la experiencia ganada en el desarrollo de este trabajo, podría plantearse la construcción de una plataforma nueva que, tomando las mejores ideas de la arquitectura del *SSCLI* para mejorar la eficiencia (entre otros aspectos), las combine con otras que mejoren u optimicen las posibilidades de ampliación dinámica del sistema, permitiendo una implementación menos compleja de capacidades reflectivas dentro de la máquina. Entre estas ideas podemos citar algunas como:

- **Establecer una correlación de estructuras de datos en memoria menos estricta**: De cara a lograr la mayor optimización posible del sistema, en *SSCLI* los objetos están ordenados en memoria con una estructura muy rígida. De esta forma, todos los objetos tienen una serie de zonas que deben estar correlativas y se hacen optimizaciones muy agresivas suponiendo la localización de determinadas partes de los objetos en lugares concretos, como ha quedado patente en el capítulo de implementación. Por tanto, a pesar de que estas decisiones se habrán tomado porque suponen una ganancia de rendimiento significativa, de cara a soportar ambos modelos computacionales sí que sería deseable no establecer unas restricciones tan fuertes en este sentido y permitir, por ejemplo, que las estructuras de datos que guardan atributos y métodos puedan crecer sin riesgo para la estabilidad del sistema, de manera que se puedan usar los mecanismos de búsqueda optimizados que se han implementado sobre una estructura de datos que pueda crecer en tiempo de ejecución.
- **Establecer menos límites estáticos para el tamaño de los elementos**: En toda la máquina virtual existen elementos cuyo tamaño está prefijado y no es posible su reconfiguración. Si bien esto nuevamente se hace por motivos de eficiencia, limita la ampliación del sistema al no poder albergar fácilmente más información dentro de los mismos.
- **Crear un soporte más explícito para realizar llamadas de forma dinámica al entorno de ejecución**: La optimización llevada en el código del *JIT* del sistema hace que éste minimice el número de accesos en tiempo de ejecución que realiza a la información existente en el resto del entorno. Dado que sobre este tipo de accesos se ha construido todo el soporte de reflexión implementado, sería interesante que la máquina implementase un soporte más completo para su realización y no los limitase artificialmente a una serie muy limitada de casos especiales (realizando incluso comprobaciones para evitar que se usen en otros casos). La implementación de este sistema demuestra que es posible acceder con seguridad al entorno de ejecución en muchos más casos de los contemplados inicialmente, por lo que la limitación no se hace por riesgos de estabilidad, sino por maximizar el rendimiento.
- **Permitir la integración de nuevas clases (con código mixto de alto y bajo nivel) más sencilla**: La creación de una nueva clase de alto nivel que haga llamadas a funciones implementadas a bajo nivel, o que parte de sus métodos estén implementados a bajo nivel, requiere la creación manual de una clase en un lenguaje de alto nivel (*C#*) y una clase a bajo nivel (*C++*) que sea su

contrapartida, representando los valores que contendría en un momento dado. El problema de este mecanismo es que el entorno exige una alineación en memoria de la clase y sus miembros exacta para el funcionamiento correcto de la misma, y en ocasiones hay que introducir miembros ficticios para evitar errores. Alguna forma de unificar el tamaño de los tipos a bajo y alto nivel o de dotar al sistema de un soporte más sencillo para este tipo de operaciones sería un avance de cara a procurar mejores posibilidades de ampliación.

- **Permitir una creación más sencilla de objetos a bajo nivel, manejados por el recolector de basura**: El sistema en su estado actual tiene un mecanismo muy complejo para permitir crear instancias de clases existentes a bajo nivel, de manera que estas clases puedan ser manejadas por el recolector de basura. Si se permitiese una creación más sencilla de estos objetos, se facilitaría la realización de operaciones dentro de la máquina.

Éstas son simplemente algunas ideas que mejorarían la implementación de la máquina de cara a permitir una mejor capacidad de ampliación sin que suponga, a nuestro juicio, una disminución del rendimiento en tiempo de ejecución elevada. De todas formas, desarrollar un sistema así es un trabajo muy complejo y de una gran extensión, que no tiene sentido abordar sin contar con un equipo y recursos adecuados, así como con un cuidadoso proceso de análisis y diseño que justifique cada uno de los pasos dados y cada módulo diseñado, para alcanzar los fines de rendimiento propuestos.

17.3 SOPORTE INTEGRAL DE LAS PRIMITIVAS DE LENGUAJES DINÁMICOS

En el prototipo presentado en esta tesis se han tratado de implementar todas las primitivas de reflexión usadas por los lenguajes dinámicos vistos. No obstante, dada la dificultad de la tarea, las restricciones temporales y el hecho de tener que demostrar la viabilidad de la idea mediante un prototipo antes de intentar desarrollarla totalmente, ha hecho que ciertos aspectos relativos a las primitivas de reflexión no hayan sido implementados aún, dejándose para un desarrollo futuro completar este trabajo y lograr así que el sistema soporte íntegramente las primitivas de reflexión. Los aspectos no implementados en este primer prototipo son:

- Metaclases que mejoren la reusabilidad y la capacidad de composición de las entidades desarrolladas al programar.
- Modificación dinámica de tipos, haciendo posible la modificación de la clase de un objeto en tiempo de ejecución y logrando así un soporte completo de reflexión estructural.
- Combinación de los tipos estáticos con los dinámicos, haciendo posible su uso conjunto dentro del mismo lenguaje de programación.
- Programación generativa en código de un lenguaje de alto nivel.
- Desarrollo de servicios específicos para la programación orientada a aspectos, como el patrón de diseño *Decorator* [GOF94], que será aplicado y modificado en tiempo de ejecución usando reflexión.

Estas características serán añadidas a la librería estándar de *SSCLI* (*BCL*) (implementando todo el soporte de bajo nivel necesario para ello), para que cualquier lenguaje pueda beneficiarse de ellas, y a un lenguaje de alto nivel específico. Combinando el uso de tipos estáticos y dinámicos se modificará el diseño de un lenguaje de la plataforma *.NET* para crear a partir del mismo un nuevo lenguaje estructuralmente reflectivo que soporte las características mencionadas, estando en estudio el empleo del lenguaje *C#* como punto de partida para esta tarea.

17.4 IMPLEMENTACIÓN DE COMPILADORES DE LENGUAJES DINÁMICOS

Dadas las aplicaciones que ya hemos mencionado de los lenguajes dinámicos y sus principales problemas (derivados de la eficiencia), el poseer ahora un sistema que logra un rendimiento superior de las primitivas dinámicas soportadas sobre el mismo, mediante el empleo de un compilador *JIT* y una máquina virtual profesional, permite desarrollar el trabajo realizado un paso más allá. Lo que se propone como continuación directa de este proyecto es la implementación de soporte de alto nivel para características dinámicas que ofrezcan una seguridad equivalente a la ofrecida por un lenguaje que cuente con tipos estáticos y al mismo tiempo incorpore la flexibilidad de los tipos dinámicos, haciendo posible la aplicación de los tipos estáticos cuando sea posible (para ganar eficiencia) y los tipos dinámicos cuando sean necesarios [Meier05].

Para ello, usando el sistema desarrollado en esta tesis, se trataría de ofrecer un soporte a alto nivel de determinadas características de los lenguajes dinámicos, obteniendo las ventajas de ambos sistemas de tipos. La plataforma para la que se hará este trabajo será la desarrollada en esta tesis, y el marco de trabajo empleado para crear código para esta plataforma y su optimización será *Phoenix* [MSRPhoenix06] de *Microsoft*. En este mismo sentido, una vez completado el soporte para las primitivas de reflexión en el sistema extendido, se podrían crear compiladores de lenguajes dinámicos, como *Python* o *Ruby,* sobre nuestro sistema, de manera que puedan aprovecharse de las características de la máquina virtual y de las nuevas funcionalidades creadas sobre la misma, además de conseguir una ejecución con un rendimiento superior, gracias a que en nuestro sistema extendido las primitivas de reflexión sobre las que se basan estos lenguajes tienen un mejor rendimiento. Por ello, es razonable implementar compiladores de estos lenguajes sobre esta máquina para comprobar hasta que punto se pueden aprovechar de todas las ventajas que ofrecería.

17.5 MIGRAR LA IMPLEMENTACIÓN AL ENTORNO COMERCIAL CLR

Dado el alto nivel de similitud existente entre el *SSCLI* y el *CLR*, motivo que ha contribuido a su elección como sistema de partida como ya hemos visto, cabe la posibilidad de tratar de migrar la solución implementada en el primer sistema al segundo. *SSCLI* es creado a partir del código del sistema comercial *CLR*, eliminando una determinada cantidad de funcionalidades, y cambiando fundamentalmente dos

subsistemas por otros más adecuados para su estudio y modificación o reemplazo: el compilador *JIT* y el recolector de basura.

Si se consiguiese trasladar la implementación hecha al sistema comercial *CLR*, adaptando el código desarrollado para trabajar con los componentes que cambien, y tomando las medidas necesarias para integrarlo con todo el sistema, entonces habremos conseguido trasladar las funcionalidades implementadas a un sistema comercial cuya distribución es muy amplia, a nivel mundial, capacitando a multitud de usuarios para que saquen partido del mismo.

Por otra parte, el *CLR* es en sí más eficiente y mucho más optimizado que el *SSCLI*, por lo que los resultados de rendimiento que pudiesen ofrecer serian probablemente mejores.

De todas formas, dado que el *CLR* es un estándar cuyo código es cerrado, sólo la propia *Microsoft* podría optar por dar este paso, ya que no es posible acceder al código del *CLR* de la misma forma y con la misma facilidad que se accede al *SSCLI*.

APÉNDICES

18 APÉNDICE A: SSCLI

18.1 INTRODUCCIÓN

En este apéndice se hará una descripción detallada de la implementación de la plataforma escogida como base: el *SSCLI* o *Rotor* de *Microsoft* [Rotor05]. Describiremos los conceptos más importantes del sistema y cómo han sido implementados en el mismo, a partir de la descripción hecha en [Stutz03]. Mediante este capítulo se pretende ofrecer pues una descripción técnica de aquellos elementos que se han modificado, mostrando como han sido construidos originalmente sin entrar en excesivos detalles técnicos, que puedan dificultar la comprensión de los conceptos que se están tratando de ilustrar en cada caso, pero con el detalle suficiente como para entender cómo se ha construido el sistema y su arquitectura general. Dado que *SSCLI* es una implementación del estándar *CLI*, en la descripción de algunos componentes aparecerán ambos términos indistintamente, ya que se ha tratado de que la descripción abarque aspectos generales del estándar además de otros aspectos de implementación más concretos.

Por último, destacar que el objetivo de este capítulo no es ofrecer una guía de las posibles modificaciones que se puedan realizar en la máquina ni una descripción exhaustiva de la arquitectura completa. Ya existe bibliografía de excelente calidad que describe la arquitectura en sí [Stutz03] y el conjunto de modificaciones posible es muy amplio y variado, por lo que se ha decidido no proceder a su descripción ya que no aporta nada significativo a esta tesis.

18.2 MODELO COMPUTACIONAL DE SSCLI

18.2.1 Descripción General de Tipos, Objetos y Componentes en SSCLI

El estándar *CLI*, y particularmente *SSCLI*, usa en toda su arquitectura un modelo computacional orientado a objetos basado en clases. Desde el punto de vista de los lenguajes de alto nivel, en el *CLI* se usará la abstracción de tipo para construir componentes, y es mediante los mismos como el programador podrá interactuar con el código de bajo nivel que implementa las diferentes partes del *CLI* (hilos, etc.) y también con elementos propios de la arquitectura del sistema (registros, espacios de direcciones, etc.). Por tanto, los tipos son el medio por el cual un programador, que trabaje con un lenguaje soportado por una implementación del estándar *CLI* (como *SSCLI*), puede interactuar con el resto de elementos de bajo nivel.

En la especificación del *CLI* aparecen tres términos relacionados con el sistema de tipos: tipo, objeto y componente. Éstos deben distinguirse y definirse correctamente antes de pasar a estudiar el resto del sistema.

18.2.1.1 TIPO

Un tipo es una especificación que describe cómo va a ser interpretado dentro del motor de *CLI* un código determinado. Los tipos son una forma de clasificar la estructura de los datos (sus atributos), las operaciones que son posibles sobre esos datos (sus métodos) y cómo se espera que estas operaciones funcionen cuando sean usadas (su implementación). El uso de tipos es un medio probado para la construcción de *software* de gran volumen, que permite clasificar adecuadamente todas las entidades que forman parte de un *software* determinado y poseen una serie de facilidades que se pueden usar para automatizar y enriquecer cada parte del proceso de programación (desde la compilación hasta la ejecución).

CLI es un entorno que trata sus tipos de forma segura, es decir, que cada objeto, variable o dato en general que pueda ser usado dentro del entorno tendrá siempre asociado un tipo durante toda su vida útil. Además, el compilador producirá código que seguirá estrictamente las reglas del sistema de tipos, permitiendo un control completo de cómo se usan. Por ejemplo, si una variable se declara como real de 32 *bits*, el compilador no debe permitir que se le asigne una cadena de caracteres, ya que de permitirlo violaría las reglas semánticas del programa. Por supuesto, este tipo de comprobaciones se hacen en muchos más puntos (comprobar si al llamar a un método M de un tipo T ese método existe, si una llamada a un método cumple con la signatura declarada del mismo, etc.).

Por otra parte, el sistema cuenta con un doble sistema de comprobación. Tanto el compilador como el entorno hacen una serie de comprobaciones de tipo por separado, de forma que sea posible evitar errores producidos por compiladores defectuosos (que no chequeen den por buena algún tipo de restricción que resulte ser errónea, permitiendo que pase a la ejecución), o bien código malintencionado que en ejecución pueda efectuar alguna operación inválida que afecte a la estabilidad del sistema.

18.2.1.2 OBJETO

[ECMA05] Según lo visto anteriormente, los tipos describen valores, estableciendo un contrato entre el tipo y todos sus posibles valores. *SSCLI* (y el *.NET Framework*) definen el concepto de *Common Type System* (*CTS*), que comprende el conjunto de tipos que usará cualquier lenguaje destinado para esta plataforma. El *CTS* define un conjunto determinado de tipos común a todos los lenguajes, que puede ser usado por cualquiera de ellos independientemente de su sintaxis. Cada lenguaje puede tener la sintaxis que desee, pero si ha de funcionar sobre la máquina, sus tipos deben pertenecer ineludiblemente al conjunto definido por el *CTS*. Como el *CTS* soporta tanto lenguajes de programación orientados a objetos (*OOP*) como lenguajes procedurales o funcionales, el *CLI* permite trabajar con dos tipos de entidades: objetos y valores.

- Los valores son datos simples, como enteros o flotantes. Cada valor tiene un tipo, que describe el tamaño del mismo para todos los lenguajes del *CLI* (tamaño que siempre será el mismo en todos ellos), el significado de cada *bit* en su representación binaria y las operaciones que pueden hacerse con él. Están pensados para representar tipos simples, que no son objetos.
- Los objetos por otra parte son más complejos que los valores:

- Cada objeto es autodescriptivo, es decir, que a partir de cualquier objeto en memoria se podrá obtener una referencia a su tipo, de la que se puede extraer su definición exacta.
- Cada objeto tiene identidad, que establece una distinción entre un objeto y cualquier otro. El contenido de la memoria en la que está guardado un objeto puede cambiar, pero la identidad de los mismos no.
- Cualquier objeto puede guardar referencias a otras entidades (otros objetos o valores).
- Todo objeto del *CLI* deriva del tipo `System.Object`.

En *SSCLI* los objetos son representados internamente en tiempo de ejecución por la clase *C++* `Object`, que es un reflejo a bajo nivel de la clase de alto nivel `Object` perteneciente a la librería de clases estándar *BCL*. Si tenemos un objeto cualquiera a alto nivel (perteneciente a un lenguaje de la plataforma) que deba ser usado por alguna operación de bajo nivel (el código que implementa el motor de ejecución), por ejemplo como uno de sus parámetros, entonces lo que recibe la operación a bajo nivel es precisamente una instancia de esa clase *C++* `Object`. Al mismo tiempo, la clase `Object` de alto nivel tiene algunas operaciones que se han de implementar a bajo nivel, por necesitar usar recursos de la máquina no accesibles directamente. Estas operaciones, marcadas como `InternallCall`, pueden no implementarse en su contrapartida de bajo nivel `Object`, sino en otra clase diferente (también de bajo nivel) denominada `COMObject`. Existe además una tercera clase, `CObjectHeader`, que representa a un objeto cuando es manipulado en el *heap* del recolector de basura. Por tanto, *SSCLI* posee 3 diferentes puntos de vista a la hora de tratar con un objeto, ya que la estructura de una instancia cualquiera está formada a nivel interno por el código de tres clases diferentes. Un usuario se encontrará, cuando trate con un objeto a alto nivel, un único tipo, del que se muestra a continuación únicamente su *interface* de operaciones y cómo algunas de ellas tienen una implementación primitiva (dentro del motor de ejecución, en las clases mencionadas):

```
[Serializable()]
public class Object
{
    // Creates a new instance of an Object.
    public Object();

    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    private extern Type InternalGetType();

    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    private extern Type FastGetExistingType();

    // Returns a String which represents the object instance.  The default
    // for an object is to return the fully qualified name of the class.
    public virtual String ToString();

    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    public extern virtual bool Equals(Object obj);

    public static bool Equals(Object objA, Object objB);

    public static bool ReferenceEquals (Object objA, Object objB);

    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    public extern virtual int GetHashCode();

    public Type GetType();

     ~Object();
```

```
    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    protected extern Object MemberwiseClone();

    private void FieldSetter(String typeName, String fieldName, Object val);

    // Gets the value specified in the variant on the field
    private void FieldGetter(String typeName, String fieldName, ref Object val);

    // Gets the field info object given the type name and field name.
    private FieldInfo GetFieldInfo(String typeName, String fieldName);
}
```

Además, aunque teóricamente se ha distinguido claramente entre valor y objeto, en la práctica el sistema de tipos del *CLI* hace que cualquier tipo (valor u objeto) descienda de `System.Object` (a diferencia de *Java*), lo que a efectos prácticos significa que es posible guardar cualquier valor en un tipo `Object`. Esto implica que los métodos disponibles en la clase `Object` mostrados anteriormente también están disponibles en los valores o tipos primitivos. Por último, es de destacar que en este código:

```
int número = 1;
Object o = número;
```

Cualquier cambio subsiguiente al valor de `o` no afecta a `número`, la primera asignación copia sólo su valor.

18.2.1.3 COMPONENTE

Los componentes [MSDNComp06] son las unidades abstractas que permiten la reutilización de código en los sistemas que se basan en ellos. En el *CLI*, los componentes permiten también la interoperabilidad entre los lenguajes que se diseñan para este sistema.

La característica más importante de los componentes en el *CLI* es su capacidad de ser empaquetados como entidades autónomas, es decir, que son unidades reemplazables en un momento dado dentro de un *software* concreto y que también podrían adaptarse a nuevas necesidades sin alterar el comportamiento esperado por la definición de su tipo. Un componente debería por tanto ser reutilizado o modificado sin necesidad de cambiar los componentes con los que éste colabora, que sólo conocerán la interfaz de operaciones que oferta a los demás.

Un componente es una abstracción más elevada que un objeto, y tiene asociados otros conceptos que definiremos a lo largo de este capítulo, pero que en general no difiere de la definición que se puede encontrar en otras plataformas similares [SunComp06]. Un objeto es un concepto que se limita a designar a cada instancia del tipo `System.Object`, mientras que un componente es una entidad más compleja que, aun siendo también un objeto, tiene asociados otros elementos como eventos, delegados, espacios de nombres, etc. de los que un objeto en sí carecería.

18.2.2 Sistema de Tipos de SSCLI

El sistema de tipos en el *CLI* tiene una doble función:

- Da a los programadores una serie de conceptos consistentes e inmutables para desarrollar aplicaciones.
- Permite que un programa puede ser comprobado en busca de incorrecciones, mejorando la seguridad, robustez y estabilidad del entorno de ejecución.

La ingeniería del *software* en general trata de asegurar que un sistema se comporta de forma correcta, y para ello éste debe cumplir exactamente cada una de sus especificaciones. Esto requiere hacer una serie de comprobaciones, pero la forma de realizarlas puede variar. Una de estas formas es la formalización matemática de programas, que asegura de forma lógica que un programa no tendrá ningún comportamiento inesperado. Otra forma es usar herramientas que chequeen automáticamente el programa, incluidas muchas veces en los entornos de desarrollo que los programadores usan. Mientras la primera opción requiere la creación de unos desarrollos muy complejos, que exigen muchos conocimientos al programador, los segundos (chequeo de modelos, monitorización en tiempo de ejecución etc.) le exigen menos conocimientos y son de una complejidad inferior, lo que permite extender más su uso. Del segundo tipo, uno de los mecanismos más extendidos es la verificación de tipos.

Un sistema que compruebe (estática o dinámicamente) si todos los tipos definidos en el mismo son usados correctamente (tal y como se han descrito en su especificación) se denomina fuertemente tipado. Este tipo de sistemas evitan código erróneo o malicioso prohibiendo las operaciones que no pueden ser verificadas como seguras en cuanto a su tipo, siguiendo una serie de reglas establecidas y definidas también en el estándar *CLI*. Cuando un componente se compila usando el *JIT* del motor de ejecución, además de producirse código nativo ejecutable también se realizan una serie de verificaciones que eviten la generación de código erróneo o malintencionado. De esta forma se establece una garantía que asegura la integridad de todo el sistema. Los beneficios concretos que se obtienen por el uso de sistemas de tipos como los descritos son:

- Detección de errores: Los tipos son usados para detectar zonas de código en las que el programador incurre en alguna operación o comportamiento incorrecto (como llamar a un método inexistente) en tiempo de compilación. Desde el punto de vista del programador es mejor que el entorno de desarrollo o el compilador detecten este tipo de errores mientras se programa y no en tiempo de ejecución.
- Mantenimiento: Los programadores usan el sistema de tipos para facilitar la refactorización de código. Para soportar nuevas características se puede cambiar la definición de un tipo y compilarla. El compilador encontrará en que partes del sistema el código ha dejado de ser válido o consistente, y el programador obtendrá una descripción de las incompatibilidades encontradas, pudiendo pues tomar las medidas oportunas para hacer que el programa vuelva a funcionar correctamente. Aunque esta técnica no es lo más aconsejable para ampliar un sistema desde el punto de vista de la ingeniería, es una práctica muy común.
- Abstracción: Los sistemas fuertemente tipados ayudan al programador a distinguir claramente entre las diferentes abstracciones presentes en un *software*. Si un método necesita un parámetro de tipo A, entonces el chequeo de tipos no permitirá pasarle a ese método elementos que no sean de tipo A o derivado de A, por lo que el usuario tendrá que distinguir perfectamente cada abstracción para evitar este tipo de errores.
- Documentación: La correcta construcción de componentes facilita la lectura de programas, ya que la propia estructura de la declaración de un tipo ofrece datos acerca de cómo se usa y del comportamiento esperado del mismo. La información obtenida de esta forma además es inmutable a los cambios, no como los

comentarios, que pueden quedarse obsoletos si no se mantiene una estricta política de sincronización. Además esta estructura permite el desarrollo de herramientas de documentación automática de código como *Javadoc* [SunJavadoc06] u otras herramientas similares disponibles para el *CLI*.

- Eficiencia: Extraer el tipo de un argumento en tiempo de ejecución podría proporcionar una información valiosa para optimizar el código y mejorar el rendimiento de un programa o reducir el espacio ocupado en memoria.
- Seguridad: Como ya se ha dicho, una buena comprobación de tipos permite que los tipos se usen sólo como se ha establecido en su diseño, evitando errores o usos malintencionados.

Por tanto, puede afirmarse que los tipos son contratos entre el programador y el entorno de ejecución, mediante los cuales el programador puede describir los requerimientos de espacio del tipo, dependencias y el comportamiento esperado. En *CLI* el contrato también abarca aspectos de bajo nivel, como el sistema usado para generar llamadas a métodos implementados a bajo nivel o mecanismos de interacción con el entorno de ejecución. Precisamente es este contrato lo que permite asegurar la integridad de un componente que se cargue en memoria en un momento dado, como se ha visto anteriormente.

Por otra parte, la existencia de un contrato permite también que aquellos componentes que desconozcan la estructura o comportamiento de otros (que por ejemplo le sean necesarios para desarrollar una determinada tarea) puedan usar las herramientas que el sistema tiene en tiempo de ejecución para determinar la composición de esos otros componentes (mediante sus metadatos) y actuar en consecuencia (usando el mecanismo de introspección ya descrito en esta tesis).

Los metadatos son la información usada para describir tipos en tiempo de ejecución, incluyendo su comportamiento y la información necesaria para cargarlo en memoria cuando vaya a ser usado. En el estándar *CLI,* los compiladores y otras herramientas relacionadas emiten metadatos usando una serie de *API* estándar que permiten leer/escribir esa información en memoria, que luego será comprimida y guardada en binario para futuros usos. Los metadatos se cargan dentro del motor de ejecución en ciertas estructuras de datos, que son trasladadas desde un medio de almacenamiento secundario adecuado a memoria principal, o bien rellenadas a medida que se vayan empleando. Sin estas estructuras, el motor de ejecución sería incapaz de averiguar la estructura de cualquier tipo en un programa ejecutable. Además, la representación de esta información en *SSCLI* está optimizada para sólo lectura (debido a que no está pensada para ser modificada en tiempo de ejecución), favoreciendo su consulta en tiempo de ejecución.

Concretamente, esta información se guarda en montículos (*heaps*) (para datos de tamaño variable, como las cadenas) o en tablas (para datos de tamaño fijo como la definición de atributos). Estas tablas o montículos son accedidos usando *tokens*, que contienen una referencia a la localización exacta del elemento buscado dentro de la tabla o *heap*. En el siguiente código se ve el número de *tokens* existente (uno para cada tipo de elemento, básicamente) y su composición:

```
// Token tags.
typedef enum CorTokenType
{
    mdtModule               = 0x00000000,
    mdtTypeRef              = 0x01000000,
    mdtTypeDef              = 0x02000000,
    mdtFieldDef             = 0x04000000,
    mdtMethodDef            = 0x06000000,
    mdtParamDef             = 0x08000000,
    mdtInterfaceImpl        = 0x09000000,
    mdtMemberRef            = 0x0a000000,
```

```
    mdtCustomAttribute      = 0x0c000000,
    mdtPermission           = 0x0e000000,
    mdtSignature            = 0x11000000,
    mdtEvent                = 0x14000000,
    mdtProperty             = 0x17000000,
    mdtModuleRef            = 0x1a000000,
    mdtTypeSpec             = 0x1b000000,
    mdtAssembly             = 0x20000000,
    mdtAssemblyRef          = 0x23000000,
    mdtFile                 = 0x26000000,
    mdtExportedType         = 0x27000000,
    mdtManifestResource     = 0x28000000,

    mdtString               = 0x70000000,
    mdtName                 = 0x71000000,
    // Leave this on the high end value. This does not correspond to metadata table
    mdtBaseType             = 0x72000000,
} CorTokenType;
```

Cuando se usan los metadatos desde el *CIL* o dentro del motor de ejecución, lo que realmente se usan son enteros de 32 *bits*, que son una combinación del puntero del *heap* donde está el objeto con el *token* que determina su tipo (un `CorTokenType`). Los metadatos se insertan directamente en el *CIL* y deben ser verificados en tiempo de ejecución por los motivos de seguridad antes mencionados. Si se permitiese la modificación de estos metadatos en tiempo ejecución directamente, se abriría la posibilidad de saltarse el sistema de seguridad del entorno de ejecución, pudiendo producirse resultados no deseados o dañinos. Esta restricción también condiciona la forma de abordar las modificaciones planteadas en esta tesis, ya que el sistema no permite la alteración de los metadatos que definen un tipo en tiempo de ejecución de una manera directa y sencilla, y por tanto no se pueden integrar en esa estructura de datos ya definida por el sistema los añadidos que se le hagan dinámicamente a un tipo dado.

Como los contratos que se hacen con los tipos sirven para que los componentes desarrollados de forma independiente a los demás puedan estudiar y usar los recursos de cualquier otro componente, entonces la información contenida en los metadatos debe ser lo más rica posible, para no limitar este tipo interacciones entre componentes.

18.2.2.1 Procesamiento del Código que Define un Componente

Aunque se describirá más adelante el proceso de compilación *JIT* que tiene lugar en *SSCLI*, se mostrará brevemente cómo el motor de ejecución trata un componente cuando éste se compila y se deja listo para ser usado. El lenguaje intermedio *CIL* es un lenguaje cuyo juego de instrucciones es neutral, es decir, que no está pensado para ninguna arquitectura particular. Este lenguaje nunca será ejecutado directamente por la máquina en *SSCLI*, sino que debe ser traducido a instrucciones nativas del microprocesador físico antes de poder ejecutar el programa en él. El siguiente código muestra un ejemplo de *CIL,* en donde se puede ver claramente cómo se integran en el programa todos los metadatos que son normalmente introducidos en el *CIL* por los compiladores. La carga de información que dependa del tipo se guía por estos *tokens*.

```
.module Hola.exe
// MVID: {8F54E3AE-EB2A-4D97-ADFA-B521A658A5E3}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x00e20000
//
// ============== CLASS STRUCTURE DECLARATION ==================
```

```
//
.class /*02000002*/ public auto ansi beforefieldinit Hello
       extends [mscorlib/* 23000001 */]System.Object/* 01000001 */
{
} // end of class Hello


// ==============================================================
// =============== GLOBAL FIELDS AND METHODS ====================
// ==============================================================


// =============== CLASS MEMBERS DECLARATION ====================
//   note that class flags, 'extends' and 'implements' clauses
//          are provided here for information only

.class /*02000002*/ public auto ansi beforefieldinit Hello
       extends [mscorlib/* 23000001 */]System.Object/* 01000001 */
{
  .field /*04000001*/ private int32 atributo
  .method /*06000001*/ public hidebysig virtual instance string  ToString() cil managed
  {
    // Code size       10 (0xa)
    .maxstack  1
    .locals /*11000001*/ init (string V_0)
    IL_0000:  ldstr      "Hello" /* 70000001 */
    IL_0005:  stloc.0
    IL_0006:  br.s       IL_0008

    IL_0008:  ldloc.0
    IL_0009:  ret
  } // end of method Hello::ToString

  .method /*06000002*/ public hidebysig static void Main(string[] prms) cil managed
  {
    .entrypoint
    // Code size       29 (0x1d)
    .maxstack  1
    .locals /*11000002*/ init (class Hello/* 02000002 */ V_0)
    IL_0000:  newobj     instance void Hello/* 02000002 */::.ctor() /* 06000003 */
    IL_0005:  stloc.0
    IL_0006:  ldloc.0
    IL_0007:  callvirt   instance string Hello/* 02000002 */::ToString() /* 06000001 */
    IL_000c:  call       void [mscorlib/* 23000001 */]System.Console/* 01000003 */
                                                ::WriteLine(string) /* 0A000002 */
    IL_0011:  ldloc.0
    IL_0012:  ldfld      int32 Hello/* 02000002 */::atributo /* 04000001 */
    IL_0017:  call       void [mscorlib/* 23000001 */]System.Console/* 01000003 */
                                                ::WriteLine(int32) /* 0A000003 */
    IL_001c:  ret
  } // end of method Hello::Main

  .method /*06000003*/ public hidebysig specialname rtspecialname instance void .ctor() cil
                                                                                  managed
  {
    // Code size       15 (0xf)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  ldc.i4.s   10
    IL_0003:  stfld      int32 Hello/* 02000002 */::atributo /* 04000001 */
    IL_0008:  ldarg.0
    IL_0009:  call       instance void [mscorlib/* 23000001 */]System.Object /* 01000001 */
                                                        ::.ctor() /* 0A000004 */
    IL_000e:  ret
  } // end of method Hello::.ctor

} // end of class Hello
```

En este código puede verse claramente cómo aparecen los *tokens* de cada elemento del programa entre comentarios, tomando la forma de un número. Si se tradujesen directamente los metadatos a direcciones o desplazamientos de memoria específicos de la arquitectura sobre la que se está ejecutando la máquina, entonces no se conseguiría la portabilidad del programa. Para evitar esto el compilador *JIT* examina los

metadatos sólo cuando es necesario usarlos, empleando los *tokens* y las tablas ya mencionadas para decidir cómo transformar esa representación intermedia en estructuras de datos y código compilado finalmente ejecutable.

18.2.2.2 VERSIONADO DE TIPOS

La evolución de los tipos es algo primordial a la hora de producir *software*. Es muy probable que, a pesar del tiempo invertido en el diseño de un programa concreto, finalmente algunos de sus tipos tengan que ser descartados o modificados para adaptarse a nuevos requisitos o solucionar problemas que surjan a medida que se vaya realizando la implementación. Ni *Java* ni *C++* ofrecen un soporte para versiones de los tipos creados en su modelo, por lo que un programador tiene que preocuparse de crear un mecanismo propio para tener esta funcionalidad. Esto puede acarrear problemas, ya que el mecanismo puede no ser completo o diferir, hasta el punto de ser incompatible, de otros mecanismos usados para el mismo propósito por otros programadores, incluso del mismo proyecto, creando un problema serio en el desarrollo. No se va a hablar aquí de mecanismos usados para hacer versiones de piezas *software*, sino que se va a describir el problema principal que causa el versionado de *software* y cómo se soluciona en el entorno *SSCLI*.

En los sistemas operativos tradicionales, una gran parte del *software* que se ejecuta sobre ellos necesita usar librerías externas, que contienen componentes y otro tipo de entidades necesarias para su correcto funcionamiento. Si esa librería se actualiza o se corrige de algún modo, entonces se genera otra versión de la misma con la nueva funcionalidad. No obstante, cuando se va a hacer uso de un recurso externo, un sistema operativo devuelve el primer elemento que encuentre que sea acorde con la descripción proporcionada, y si no se tiene cuidado con el lugar donde se introduce cada versión, el *software* accidentalmente puede usar una versión errónea, alterando la ejecución del programa y causando errores. Si en vez de eso reemplazamos la librería antigua por la nueva, es posible que haya otro *software* que necesite hacer uso de la versión anterior de esa librería, al no estar preparado para los cambios introducidos (es decir, que sólo estará pensado para trabajar con la versión anterior), con lo que impediríamos la ejecución de otro programa. Este problema, originado por el criterio de selección empleado por el sistema operativo para localizar una librería concreta, puede ser extendido al criterio empleado por el entorno de ejecución para seleccionar una versión concreta de un componente. Toda esta problemática es ocasionada por la limitada información de la que se dispone de cada elemento, que impediría hacer una selección correcta en cada caso. No obstante, este problema se ha solucionado en el *CLI*, ya que este entorno especifica una serie de reglas por las que un componente se identifica, usando para ello 4 elementos:

- **Número de versión**.
- **Información de internacionalización (*locale*)**.
- **Una clave criptográfica pública (*strong name*)**.
- **El nombre concreto del componente**.

Aunque está claro que es el programador el que finalmente determina cómo un tipo evoluciona (generando las sucesivas versiones del mismo) y cómo usará dicho tipo las evoluciones de otros componentes que necesite en su implementación (al ser un proceso altamente dependiente de su diseño), esta información extendida permite la localización de un componente y una versión específica del mismo de una forma mucho más efectiva. Además, el entorno incluye reglas concretas que describen el proceso por

el cual un componente es evaluado como posible candidato para ser cargado en memoria y utilizado.

18.2.2.3 DESCRIPCIÓN DE TIPOS

La autodescripción de tipos del *CLI* que se ha mencionado anteriormente es uno de los aspectos más importantes de su diseño. El "marcado" de cada componente con metadatos que lo describan completamente es la base para muchas de las funcionalidades que el *CLI* ofrece. Esta capacidad permite, como ya se ha mencionado, posponer decisiones de compilación, enlazado y carga en memoria si es necesario, mejores mecanismos de versionado y una evolución del *software* a construir más sencilla. Para ejemplificar el modo en que el empleo de metadatos proporciona los beneficios que se han mencionado anteriormente, mostraremos el siguiente ejemplo:

Este código declara una clase `Entero` en *C++*:

```
#include <iostream>
using namespace std;

class Entero
{
  private:

        int valor;

  public:

        Entero(){
                valor = 0;
        }

        int getEntero (){
                return valor;
        }

        void setEntero (int valor) {
                this->valor = valor;
        }
};

void main ()
{
        Entero e;

        e.setEntero(5);
        cout << "Valor : "<< e.getEntero();
}
```

Y el siguiente fragmento muestra la traducción de un método del mismo (`getEntero`) a código ensamblador *x86*:

```
?getEntero@Entero@@QAEHXZ (public: int __thiscall Entero::getEntero(void)):
  00000000: 55                 push        ebp
  00000001: 8B EC              mov         ebp,esp
  00000003: 81 EC CC 00 00 00  sub         esp,0CCh
  00000009: 53                 push        ebx
  0000000A: 56                 push        esi
  0000000B: 57                 push        edi
  0000000C: 51                 push        ecx
  0000000D: 8D BD 34 FF FF FF  lea         edi,[ebp+FFFFFF34h]
  00000013: B9 33 00 00 00     mov         ecx,33h
  00000018: B8 CC CC CC CC     mov         eax,0CCCCCCCCh
  0000001D: F3 AB              rep stos    dword ptr [edi]
  0000001F: 59                 pop         ecx
```

```
  00000020: 89 4D F8           mov         dword ptr [ebp-8],ecx
  00000023: 8B 45 F8           mov         eax,dword ptr [ebp-8]
  00000026: 8B 00              mov         eax,dword ptr [eax]
  00000028: 5F                 pop         edi
  00000029: 5E                 pop         esi
  0000002A: 5B                 pop         ebx
  0000002B: 8B E5              mov         esp,ebp
  0000002D: 5D                 pop         ebp
  0000002E: C3                 ret
```

Los siguientes dos fragmentos de código muestran lo mismo que se ha hecho en *C++*, pero programado en el lenguaje *C#*, sobre la plataforma *CLI*:

```
using System;

class Entero
{
        private int valor;

        public Entero(){
                valor = 0;
        }

        public int getEntero (){
                return valor;
        }

        public void setEntero (int valor) {
                this.valor = valor;
        }

        public static void Main ()
        {
                Entero e = new Entero();

                e.setEntero(5);

                Console.WriteLine("Valor : " + e.getEntero());
        }

};
```

```
.module Entero.exe
// MVID: {FE3198DF-4D51-4047-8E8F-217245DE0F6D}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x00e20000
//
// ============== CLASS STRUCTURE DECLARATION ==================
//
.class private auto ansi beforefieldinit Entero
       extends [mscorlib]System.Object
{
} // end of class Entero


// =============================================================
// ============== GLOBAL FIELDS AND METHODS ====================
// =============================================================


// ============== CLASS MEMBERS DECLARATION ====================
//   note that class flags, 'extends' and 'implements' clauses
//          are provided here for information only

.class private auto ansi beforefieldinit Entero
       extends [mscorlib]System.Object
{
  .field private int32 valor
  .method public hidebysig specialname rtspecialname instance void  .ctor() cil managed
```

```
  {
    // Code size       14 (0xe)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ldarg.0
    IL_0007:  ldc.i4.0
    IL_0008:  stfld      int32 Entero::valor
    IL_000d:  ret
  } // end of method Entero::.ctor

  .method public hidebysig instance int32 getEntero() cil managed
  {
    // Code size       11 (0xb)
    .maxstack  1
    .locals init (int32 V_0)
    IL_0000:  ldarg.0
    IL_0001:  ldfld      int32 Entero::valor
    IL_0006:  stloc.0
    IL_0007:  br.s       IL_0009

    IL_0009:  ldloc.0
    IL_000a:  ret
  } // end of method Entero::getEntero

  .method public hidebysig instance void setEntero(int32 valor) cil managed
  {
     // Code size       8 (0x8)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  stfld      int32 Entero::valor
    IL_0007:  ret
  } // end of method Entero::setEntero

  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
     // Code size       40 (0x28)
    .maxstack  2
    .locals init (class Entero V_0)
    IL_0000:  newobj     instance void Entero::.ctor()
    IL_0005:  stloc.0
    IL_0006:  ldloc.0
    IL_0007:  ldc.i4.5
    IL_0008:  callvirt   instance void Entero::setEntero(int32)
    IL_000d:  ldstr      "Valor : "
    IL_0012:  ldloc.0
    IL_0013:  callvirt   instance int32 Entero::getEntero()
    IL_0018:  box        [mscorlib]System.Int32
    IL_001d:  call       string [mscorlib]System.String::Concat(object,
                                                                object)
    IL_0022:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0027:  ret
  } // end of method Entero::Main

} // end of class Entero
```

Nótese como la versión *C++* de este código origina un código nativo que tiene incluido el desplazamiento (*offset*) de un atributo de la misma desde la dirección de memoria donde la clase ha sido cargada. Si posteriormente añadimos un nuevo atributo a esta clase, los *offsets* cambiarán y el código nativo generado que use esta clase ya no será válido tal y como está, de ahí que cada cambio en un programa de este tipo necesite una recompilación. Por otra parte, si se trata de portar el código a otra arquitectura con otro microprocesador, el código también fallará, ya que el juego de instrucciones usado será diferente. La versión *C++* de la clase `Entero` también carece de información sobre si misma, por lo que ninguna herramienta podrá emplear el código para extraer información acerca de dicha clase, salvo que explícitamente se genere dicha información por motivos de depuración.

Por otra parte, la versión *C#* de esta clase no se basa en *offsets* para acceder a

los elementos de la misma. En vez de ello, se emite un *token* de metadatos para el atributo `Entero::valor`. Cuando el tipo se carga, estos *tokens* sirven como puntos de búsqueda que determinan dónde está localizado el valor de cada atributo dentro de la memoria que ocupa dicho tipo. Si el tipo `Entero` se amplia de la misma forma mencionada anteriormente, como ahora el mecanismo empleado por el *CLI* depende de nombres y no de *offsets*, el código que emplee esta nueva clase `Entero` modificada seguirá funcionando correctamente. Claramente, el modelo propuesto por *C#* es mucho más adaptable que el propuesto por *C++*, ya que soporta mejor los cambios que se puedan realizar a un tipo.

18.2.2.4 VALUE TYPES

Como ya se mencionó anteriormente, los *value types* [MSDNTypes06] son la abstracción del *CLI* para representar los datos que contiene un tipo de un programa. Sin éstos los componentes no podrían tener contenido alguno, ya que cualquier tipo complejo definido acaba estando ineludiblemente formado por una colección de *value types*, directa o indirectamente. Se entiende por *value type* todo *byte*, carácter, entero (cualquier tamaño), número en punto flotante (cualquier precisión), números decimales, valores enumerados, booleanos, etc. Todo *value type* es representado por "una secuencia de bits" [ECMA05]. Adicionalmente, el sistema posee los llamados *reference types*, que guardan referencias a los *value types*. Ambas clases derivan de `System.Object`, lo que permite obtener la representación de todo valor.

Además, todo *value type* puede funcionar como atributo, parámetro, tipo de retorno de un método o como variable. Cuando un *value type* forma parte de la definición de un objeto o bien está dentro de una *array*, entonces el valor del mismo reside dentro del espacio reservado para el objeto en el *heap*. Cuando es declarado como una variable o parámetro, entonces es alojado en la pila. Si un *value type* se pasa como parámetro a un método, es una copia y no la dirección del mismo la que es creada y enviada a la implementación de dicho método por defecto (paso por valor). Varios *value types* pueden ser agrupados usando la palabra reservada `struct` o su valor puede ser restringido a una serie de valores concretos creando un tipo enumerado (`enumerate`). En la figura 18.1 se muestran los diferentes tipos del sistema y su jerarquía:

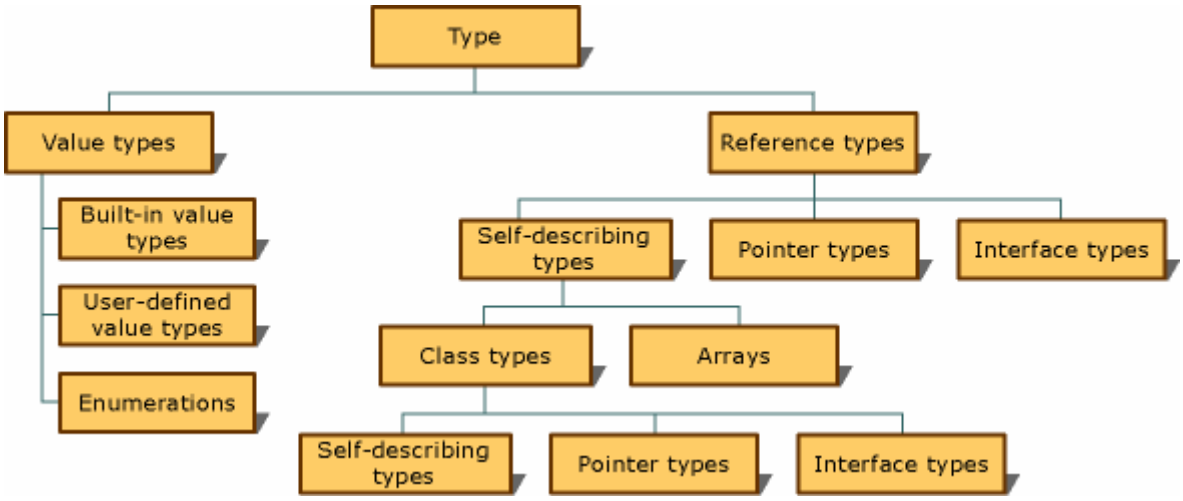

Figura 18.1: Jerarquía de tipos en *SSCLI* [MSDNTypes06]

A la hora de trabajar con *value types*, hay que tener en cuenta un par de

aspectos: la conversión de tipos y la capacidad para convertir uno de esto tipos en un *reference type*, operación denominada *boxing,* que mencionaremos a continuación.

BOXING/UNBOXING

Esta operación ocurre cuando un programador desea usar un *value type* de una forma consistente con los *reference types*. En ese caso el *value type* se somete a una operación llamada *boxing*, mediante la cual automáticamente se creará un *referente type* cuyo valor sea el del *value type*, y será ese *reference type* el que se emplee en vez del original. Por supuesto, la operación contraria (*unboxing*) también es posible.

El lenguaje *CIL* tiene un *opcode* específico para ambas operaciones (`box, unbox`). Para que esto sea posible, cualquier *value type* debe tener asociado un *reference type* que le corresponderá cuando se usa la operación de *boxing*, obteniendo lo que se llama *boxed type*. Además este *boxed type* no es accesible para el programador, y es generado automáticamente por el *CLI* cuando éste detecta que es necesario en una operación.

CONVERSIÓN DE TIPOS

La necesidad de convertir un tipo, que tiene un formato y tamaño determinado, a otro tipo diferente (un flotante a un entero, por ejemplo) es muy frecuente. Como este tipo de conversiones no pueden hacerse a la ligera, es necesario establecer una serie de reglas que especifiquen cómo a partir de una valor x de un tipo T1 se podrá obtener un valor x' de un tipo T2, cuyo significado sea equivalente al original en el nuevo tipo. Los compiladores usan estas reglas cuando en un programa se hace una operación que requiera una conversión.

Existen dos tipos de conversión. En el primer caso, un valor cuyo tipo tiene un tamaño determinado es asignado a otro valor cuyo tipo tiene un tamaño mayor, como ocurriría cuando se intenta asignar un entero de 32 *bits* a un entero de 64 *bits*. El segundo caso es justo el contrario: el tamaño del tipo del valor al que se le va a asignar otro valor es menor que el del tipo de ese otro valor. En el primer caso, al no incurrir en posibles pérdidas de información, no se presentan problemas. No obstante, en el segundo caso sí se pueden dar situaciones en las que se puede incurrir en pérdida de precisión o de información y errores inesperados, ya que incluso el valor finalmente asignado puede ser diferente del original debido a una mala conversión. Aunque el *CLI* es capaz de detectar y notificar errores en el segundo tipo de conversiones, si se desea que de todas formas se produzca la conversión asumiendo el error de precisión, entonces el programador debe hacer una conversión explícita mediante un *cast*.

18.2.2.5 REFERENCE TYPES

Los *reference types* se comportan de acuerdo al estado que se guarda de los mismos en el *heap*. Hay tres tipos de *reference types* en el *CLI*: objetos (ya descritos), *interfaces* y punteros encapsulados. Una referencia es en sí una pequeña sección de memoria que apunta a la localización de lo que guarda realmente el *reference type*. Este concepto es similar a un puntero tradicional, pero con las siguientes ventajas:

- Las referencias son seguras en cuanto a tipo: Una instancia no puede ser asignada a una referencia a menos que esa asignación sea compatible, es decir, que el tipo

de la instancia debe ser el indicado en la referencia o bien derivar de él.

- Las referencias no pueden ser asignadas incorrectamente: Una referencia no puede estar apuntando a una localización de memoria que no esté ocupada por un objeto del tipo que indica la misma o compatible. Tampoco es posible hacer que una referencia apunte a una zona arbitraria de memoria. En resumen, es imposible que una referencia tenga un valor sin sentido. Una referencia pues sólo podrá tener dos tipos de valores: un objeto de un tipo correcto o `null`.
- Las referencias no se liberan: Si una referencia apunta a un objeto, ese objeto no puede (y no tiene) que ser liberado. El recolector de basura se ocupará de liberar la memoria no usada.
- La memoria de una referencia no es accesible: El valor contenido en una referencia (la dirección física del objeto al que designa) no es accesible para el programador.

Por último, otra de las principales diferencias entre *value types* y *referente types* es que mientras que a los primeros se les reserva un espacio en memoria cuando son declarados, los segundos necesitan una reserva explícita de espacio mediante una sintaxis propia del lenguaje (`new` en *C#*, por ejemplo).

18.2.2.6 INTERFACES

Un *interface* es un contrato por el cual cualquier tipo que lo implemente se compromete a tener un comportamiento determinado. Es una definición de tipo estricta, en la que cada uno de los comportamientos que contiene debe ser implementado en todas las clases que declaren la implementación de este *interface*. La principal utilidad de un *interface* es la capacidad de dividir tipos en categorías según implementen o no las funcionalidades del *interface*. Por ejemplo, si un tipo complejo desea poder comparar instancias de si mismo entre ellas, debe implementar el *interface* `IComparable`, que definirá la funcionalidad necesaria para hacer dicha comparación, y entraría dentro de la categoría de objetos que se pueden comparar.

Los *interfaces* pueden contener diferentes tipos de miembros, ya sean propiedades, métodos o eventos. Como ejemplo, mostramos el *interface* anterior:

```
// The IComparable interface is implemented by classes that support an
// ordering of instances of the class. The ordering represented by
// IComparable can be used to sort arrays and collections of objects
// that implement the interface.
public interface IComparable
{
      // Interface does not need to be marked with the serializable attribute
      // Compares this object to another object, returning an integer that
      // indicates the relationship. An implementation of this method must return
      // a value less than zero if this is less than object, zero
      // if this is equal to object, or a value greater than zero
      // if this is greater than object.
      int CompareTo(Object obj);
}
```

La posibilidad de incluir en un *interface* algo más que la simple definición de métodos hace que el *CLI* permita más flexibilidad en su uso que otras plataformas, como *Java*. Así, un *interface* puede ser expresado de una forma más sencilla con lenguajes del *CLI*. El siguiente código define una clase que funcionaría como un *interface* en *C++*:

```
class PuertoSerie
```

```
{
    public:

        //Propiedad que indica la tasa de baudios de trasmision.
        virtual int getBaudios()=0;
        virtual void setBaudios(int valor)=0;

        //Propiedad que indica el control de flujo.
        virtual bool getControlFlujo()=0;
        virtual void setControlFlujo (int valor)=0;

        //Funcionamiento.
        virtual void send(int datos)=0;
        virtual int read()=0;

        //Notificación.
    public:
        class Callback
        {
        public:
                //Llamado cuando los datos estén listos para ser leidos.
                virtual void onDataReady() = 0;
        };

        virtual void registerListener(const Callback& listener) = 0;
        virtual void removeListener(const Callback& listener) = 0;
};
```

Y esto es el mismo *interface* definido en *C#*:

```
public interface PuertoSerie
{
        public int Baudios
        {
                get;
                set;
        }
        public bool ControlFlujo
        {
                get;
                set;
        }
        public void send (int datos);
        public int read ();

        public delegate void DataReadyDelegate();
        public event DataReadyDelegate OnDataReady();
}
```

Puede verse que el código de *C#* es más claro, ya que es posible examinar el *interface* e interpretar claramente su estructura extrayendo las propiedades métodos y eventos fácilmente. Además, la clase que funciona como *interface* en *C++* realmente no es un *interface* (ya que estas entidades no existen en el lenguaje), sino una clase cuyos métodos son virtuales puros y necesitan ser implementados por cualquier clase que herede de la vista, "emulando" así el concepto de *interface*.

18.2.2.7 Delegados y Punteros Gestionados (*Managed Pointers*)

Los delegados (*delegates*) y los punteros gestionados (*managed pointers*) son ambos punteros encapsulados, es decir, tipos referencia con una información extra en su definición que les permite adquirir características especiales dentro del *CLI*. Un ejemplo claro de este tipo de elementos son los punteros a funciones, que son soportados como

un elemento de primer orden dentro del *CLI*. Estos punteros se pueden referir a un método cualquiera de un objeto, pero no contendrán información suficiente para localizar el código del método y de los datos del objeto asociado al mismo. Para solventar este problema, permitir el uso de eventos y poder pasar funciones como parámetros a métodos, se crearán los delegados (*delegates*).

Los delegados son la versión orientada a objetos de los punteros a función. Contienen un puntero a un método y una referencia a una instancia de un objeto específico dentro de su definición. Al poseer ambas cosas, se pueden consultar los metadatos del objeto para asegurar que la llamada es segura en tiempo de ejecución, entre otras aplicaciones. Los punteros gestionados (*managed pointers*) son un concepto similar. Hay ciertas ocasiones que, en utilidades muy especificas, es necesario contar con punteros al estilo "tradicional", que se refieran a una dirección de memoria concreta. El *CLI* no permite el uso de este tipo de punteros directamente, ya que de hacerlo no podría asegurar la seguridad de las operaciones. Para resolver este problema, los punteros gestionados guardan, además del propio puntero, información de tipo que permite al *JIT* comprobar que este puntero se va a usar correctamente y actuar en caso de que se detecte alguna incongruencia. La clave de estos mecanismos es que son opacos, ya que sus datos internos no son accesibles, por lo que el motor de ejecución puede garantizar su uso correcto sin que el usuario tenga que hacer ninguna operación adicional.

18.2.2.8 Igualdad de Tipos y Acceso a Bajo Nivel

Identidad e Igualdad de Tipos

No debe confundirse la identidad de un tipo con la igualdad entre tipos, siendo ambos conceptos imprescindibles en la implementación del *CLI*. La identidad es una propiedad que depende de la localización en memoria de un objeto, es decir, la dirección de memoria en la que se guardan los datos de un objeto es la que determina su identidad. En cambio, la información que un objeto contiene (su estado), es precisamente lo que define si dos objetos son iguales. Por tanto, la identidad está definida por la dirección de memoria en la que están guardados los datos de un objeto, mientras que la igualdad es función de los propios datos. Si dos objetos son idénticos entonces son iguales, pero esto no tiene porque ocurrir a la inversa.

Dentro del *CLI* se contemplan estos conceptos. La identidad se comprueba mediante el método `ReferenceEquals` de la clase `Object`, mientras que la igualdad es comprobada por el método `Equals` de la misma clase. No obstante, la igualdad es algo que depende enormemente la de la definición de una clase y de la aplicación, por lo que es casi imprescindible la redefinición del cuerpo de ese método para su adaptación a las necesidades concretas dentro de dicha aplicación.

Acceso a Bajo Nivel

El *CLI* está diseñado para tener un sistema de tipos consistente, que represente todas las abstracciones de la arquitectura *hardware* sobre la que se ejecuta, además de añadir nuevas abstracciones que la amplíen. En esta sección se describirá brevemente la forma por la cual los tipos definidos en el *CLI* son trasladados a la arquitectura *hardware* sobre la que se ejecuta dicho sistema, además de describir cómo se accede a otros servicios necesarios para el funcionamiento del motor de ejecución y que son ofrecidos por la arquitectura o sistema operativo.

Tipos Básicos (*Built-In Types*):

Los tipos básicos son la forma más simple de interoperabilidad entre tipos entre los distintos lenguajes del *CLI*. Son directamente utilizables y entendibles por el motor de ejecución independientemente de los tipos que defina un lenguaje de alto nivel, ya que poseerá tipos equivalentes para cualquiera de los que pueda definir uno de estos lenguajes. Por ejemplo, el tipo `System.Int32` representa un entero de 4 *bytes* con signo. Estos tipos son normalmente trasladados directamente a tipos nativos de la arquitectura sobre la que se ejecuta el *CLI*, utilizando una lista que contiene e identifica cada tipo y es accesible sólo internamente. El método que se muestra a continuación es el encargado de usar esta lista para la conversión:

```
inline static CorInfoType toJitType(CorElementType eeType) {

    static const BYTE map[] = {
        CORINFO_TYPE_UNDEF,
        CORINFO_TYPE_VOID,
        CORINFO_TYPE_BOOL,
        CORINFO_TYPE_CHAR,
        CORINFO_TYPE_BYTE,
        CORINFO_TYPE_UBYTE,
        CORINFO_TYPE_SHORT,
        CORINFO_TYPE_USHORT,
        CORINFO_TYPE_INT,
        CORINFO_TYPE_UINT,
        CORINFO_TYPE_LONG,
        CORINFO_TYPE_ULONG,
        CORINFO_TYPE_FLOAT,
        CORINFO_TYPE_DOUBLE,
        CORINFO_TYPE_STRING,
        CORINFO_TYPE_PTR,           // PTR
        CORINFO_TYPE_BYREF,
        CORINFO_TYPE_VALUECLASS,
        CORINFO_TYPE_CLASS,
        CORINFO_TYPE_CLASS,         // VAR (type variable)
        CORINFO_TYPE_CLASS,         // MDARRAY
        CORINFO_TYPE_BYREF,         // COPYCTOR
        CORINFO_TYPE_REFANY,
        CORINFO_TYPE_VALUECLASS,    // VALUEARRAY
        CORINFO_TYPE_INT,           // I
        CORINFO_TYPE_UINT,          // U
        CORINFO_TYPE_DOUBLE,        // R

        CORINFO_TYPE_PTR,           // FNPTR
        CORINFO_TYPE_CLASS,         // OBJECT
        CORINFO_TYPE_CLASS,         // SZARRAY
        CORINFO_TYPE_CLASS,         // GENERICARRAY
        CORINFO_TYPE_UNDEF,         // CMOD_REQD
        CORINFO_TYPE_UNDEF,         // CMOD_OPT
        CORINFO_TYPE_UNDEF,         // INTERNAL
        };

    _ASSERTE(sizeof(map) == ELEMENT_TYPE_MAX);
    _ASSERTE(eeType < (CorElementType) sizeof(map));
        // spot check of the map
    _ASSERTE((CorInfoType) map[ELEMENT_TYPE_I4] == CORINFO_TYPE_INT);
    _ASSERTE((CorInfoType) map[ELEMENT_TYPE_VALUEARRAY] == CORINFO_TYPE_VALUECLASS);
    _ASSERTE((CorInfoType) map[ELEMENT_TYPE_PTR] == CORINFO_TYPE_PTR);
    _ASSERTE((CorInfoType) map[ELEMENT_TYPE_TYPEDBYREF] == CORINFO_TYPE_REFANY);
    _ASSERTE((CorInfoType) map[ELEMENT_TYPE_R] == CORINFO_TYPE_DOUBLE);

    return((CorInfoType) map[eeType]);
}
```

Como puede verse, además de los tipos básicos tradicionales de otros lenguajes, existen otros tipos con una mayor carga semántica, como `OBJECT`, *arrays* o *strings*.

Además el *CLI* también tiene un soporte completo para punteros de diferentes clases mediante los tipos `BYREF`, `PTR` o `REFANY`, aunque, como ya se ha mencionado, el empleo de estos punteros para manipular direcciones de memoria u operaciones similares tiene muchas restricciones de uso, para asegurar que su empleo es seguro y no causará fallos en la ejecución.

Recursos:

Otro problema resuelto por el diseño del *CLI* es cómo acceder a recursos del sistema operativo desde su implementación, ya que un sistema operativo proporciona recursos y abstracciones que el *CLI* puede necesitar usar. Por ejemplo, puede suministrar un nombre especial o *handle* para acceder a cierto recurso, usando un *API* especial pensado para ello, pero el *CLI* no debe tener conocimiento de dicho *handle* ni del recurso concreto en sí, para no vincularse con la estructura de ese sistema operativo concreto y comprometer así su portabilidad. Ante casos como éste, se emplean *Wrapper Classes* o clases envoltorio que permitan hacer las llamadas a este tipo de abstracciones, encapsulando estas llamadas a código propio de la arquitectura.

18.2.3 Los Ensamblados (Assemblies)

En esta sección se describirá más en detalle cómo el entorno de ejecución *SSCLI* implementa y trata los ensamblados. Los ensamblados son un mecanismo usado para empaquetar los tipos, de forma que éstos puedan ser transportados de una máquina a otra y ser reconstruidos en la máquina de destino de forma correcta y segura. Teóricamente, al menos en la versión usada para implementar el prototipo presentado en esta tesis, no es posible encontrar tipos fuera de los ensamblados, por lo que la información que contienen los ensamblados de una aplicación describe todos los tipos que esa aplicación va a tener en tiempo de ejecución. Esta información puede ser sintetizada en tiempo de ejecución, pero es más común encontrarla guardada en memoria secundaria, de forma que pueda ser transferida a otras máquinas más fácilmente y no suponga un coste mayor en la ejecución de un programa. A continuación se describirán las principales características de los ensamblados del estándar *CLI*, aunque particularizándolas para *SSCLI*. Estas características son:

- Capacidad autodescriptiva.
- Independencia de plataforma.
- Asociación con nombres que permiten operaciones avanzadas.
- Capacidad para modificar el procedimiento de carga.
- Validación.

18.2.3.1 CAPACIDAD AUTODESCRIPTIVA

Como ya se ha mencionado, los ensamblados contienen los "planos" de un conjunto determinado de tipos en forma de metadatos y *CIL*. A estos conjuntos de tipos se les denomina módulos. Por tanto, un módulo es un fichero que contiene la estructura y el comportamiento para alguno o todos los tipos y/o recursos que un ensamblado

contiene. Un ensamblado siempre contiene por lo menos un módulo, pero puede contener varios si es necesario para lograr una mayor flexibilidad. Los tipos que se guardan en un ensamblado se representan dentro de los metadatos como redirecciones a los módulos que los contienen, es decir, no pueden existir tipos sin los módulos. Los contenidos de un ensamblado pueden estar repartidos en varios ficheros.

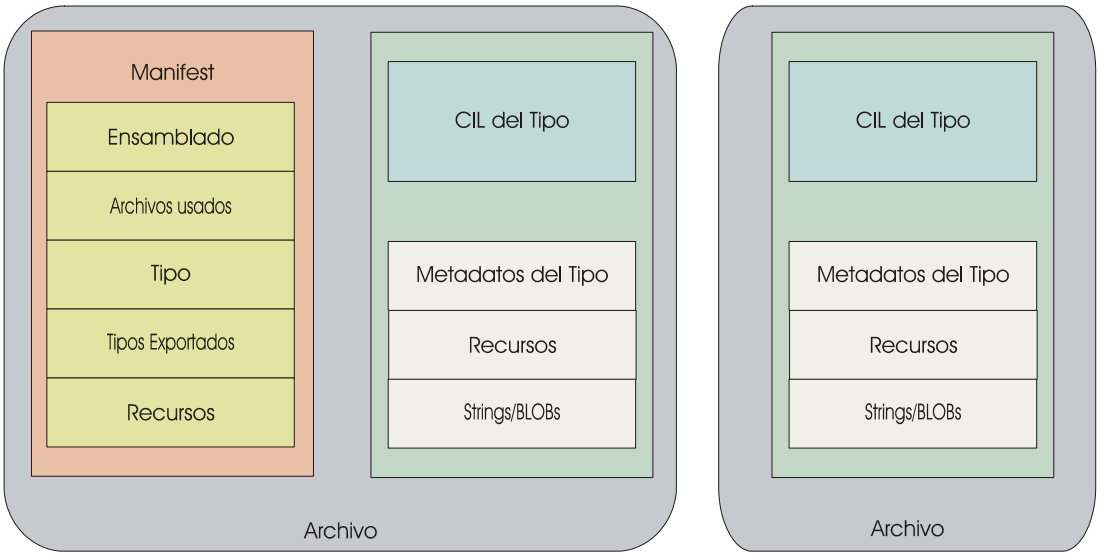

Figura 18.2: Composición de un *assembly*

En la figura 18.2 se puede ver cómo los ensamblados también pueden contener recursos, es decir datos pasivos que no son ni ejecutables ni metadatos. Los recursos son típicamente empaquetados como parte de un ensamblado para integrarse dentro de su espacio de nombres y espacio lógico. Otro elemento contenido por los ensamblados son los manifiestos (*manifest*), que describen el contenido del ensamblado. El manifiesto contiene el nombre compuesto del ensamblado, los tipos públicos que exporta y aquéllos que necesita importar de otros ensamblados. Estos manifiestos se construyen cuando el código fuente se compila satisfactoriamente y pueden ser inspeccionados mediante la herramienta *metainfo*.

18.2.3.2 LOCALIZACIÓN DE *ASSEMBLIES*

Si un ensamblado está guardado en memoria secundaria, entonces será posible cargarlo en diferentes máquinas independientemente de su arquitectura, garantizando su portabilidad, al usar un formato común. No obstante, este formato común existente en memoria secundaria es bastante diferente del formato que usaría el mismo ensamblado en memoria principal. En disco, *SSCLI* usa un formato de fichero ejecutable llamado *Microsoft Portable Executable* (*PE*) [MicrosoftCOFF06]. Esta imagen de archivo incluso no tiene porque existir solamente en disco para que *SSCLI* pueda usarla, también se admiten imágenes generadas dinámicamente o cargadas en memoria principal. Los archivos *PE* son bastante simples y una vez creados es posible obtener fácilmente varias cabeceras (por ejemplo una cabecera *Win32* (cuyo tipo es `IMAGE_NT_HEADERS`) o las cabeceras *COR* (de tipo `IMAGE_COR20_HEADERS`)) e instancias a objetos que contienen metadatos (instancias de tipo `IMDInternalImport`). Otros datos contenidos son el punto de entrada, número de versión, los segmentos donde comienzan los metadatos y recursos, etc.

Existen sólo dos clases de tipos dentro de un ensamblado: públicos (expuestos

para ser usados por otros ensamblados) e internos (sólo usados por tipos contenidos dentro del mismo ensamblado). Como ya se ha mencionado, para facilitar la adaptabilidad la vinculación entre elementos en ejecución se hace mediante el uso de nombres simbólicos y no direcciones u *offsets*. Por ello, cuando un tipo usa otro tipo, lo localiza usando su nombre (obtenido como combinación del nombre del ensamblado con el nombre del tipo usado en sí, incluyendo un prefijo adicional (su espacio de nombres)) si está presente.

En cuanto a los nombres de los ensamblados, *SSCLI* no añade nada nuevo a lo definido por el *CLI*. Un nombre completo de ensamblado está compuesto por cuatro partes, de las cuales todas menos el nombre base son opcionales. El nombre completo está compuesto por la concatenación del nombre del fichero sin extensión, número de versión mayor, número de versión menor, número de *build* y número de revisión, todos ellos separados por puntos. Tras estos elementos se incluye una referencia a la cultura asociada al ensamblado (que puede ser útil para interpretar cadenas y otros elementos contenidos en los recursos del mismo), usando una abreviatura de dos letras. Por último, una clave pública (abreviada o no) que identifica al desarrollador del ensamblado puede ser también incluida, para dar soporte a los llamados *strongnames* criptográficos y proporcionar así un nivel de seguridad mayor, al identificar la fuente del ensamblado unívocamente y poder así controlar la fiabilidad del origen del mismo.

18.2.3.3 ENLAZADO DE ENSAMBLADOS

Normalmente los ensamblados son cargados sólo cuando son necesarios, aumentando de esta forma el rendimiento del sistema al evitar la carga de elementos que nunca serán usados. Si una aplicación apenas usa un método o un recurso, el ensamblado que lo contiene puede no ser cargado en algunas ejecuciones. Una aplicación puede tener referencias a ensamblados que pueden no llegar a ser cargados.

Cuando se hace la llamada a un método dentro de un ensamblado, el primer paso a la hora de enlazarlo es decidir que versión del ensamblado que contiene el tipo que tiene ese método se usa. Para determinar esto, el entorno de ejecución consulta el manifiesto asociado al ensamblado que hace la llamada, donde se podrá encontrar la versión exacta que es necesaria en la parte correspondiente a referencias externas. Una vez obtenida, el *SSCLI* localiza el ensamblado usando un servicio llamado *fusion*. Mediante este servicio se determina si ya existe un ensamblado apropiado cargado en memoria actualmente (ningún ensamblado se carga más de una vez en memoria dentro del mismo dominio de aplicación, lo que proporciona un uso más eficiente de los recursos). Si no lo está, pero el ensamblado a cargar posee un *strongname* completamente especificado, entonces se consulta el *Global Assembly Cache* (*GAC*), del que se hablará más en detalle posteriormente. Si este ensamblado es localizado en el *GAC*, entonces se carga. Si no, el entorno de ejecución busca en los lugares especificados por la propiedad *codebase* en sus archivos de configuración, que especifican localizaciones donde pueden estar los ensamblados. Si no existen este tipo de elementos, entonces se busca en los elementos *appbase* como último recurso, que especifican un conjunto de localizaciones dentro del sistema de ficheros. Por defecto, *appbase* apunta al mismo directorio base relativo de la aplicación.

Como ya se ha mencionado, el *CLI* soporta el uso de firmas criptográficas como un medio para la identificación de ensamblados. La presencia del atributo `AssemblyKeyFileAttribute` en los metadatos del mismo identifica que dicho ensamblado posee un *strongname*, y este atributo es el que se usa a la hora de cargarlo, para asegurarse de que se carga exactamente aquello a lo que se hace referencia.

El mecanismo de verificación requiere acceso a las partes públicas y privadas de la clave criptográfica en el momento en el que el ensamblado se construye. La parte pública forma parte del nombre del ensamblado, y se calcula un código *hash* de los metadatos

del mismo usando la clave privada, insertándolo en el propio ensamblado. Cuando el ensamblado se carga, el motor de ejecución usa la parte pública para extraer el código *hash,* comparándolo contra el resultado de la generación directa de otro código *hash* de los metadatos, probando así que el productor del ensamblado tiene acceso a la clave privada. Este mecanismo no es del todo seguro, ya que ciertas partes del *SSCLI* no realizan este chequeo; se da pues la circunstancia de que actualmente ningún ensamblado construido como parte del *SSCLI* puede ser completamente seguro en cuanto a su origen.

18.2.3.4 COMPARTIR ENSAMBLADOS

El *GAC*, como ya se ha dicho, es parte del camino de búsqueda de ensamblados. *SSCLI* soporta la ejecución de varias versiones del *CLI* en una misma máquina, y el *GAC* se implementa como un subdirectorio dentro de la distribución. Este *GAC* funciona como una librería a nivel de máquina que guarda ensamblados para cualquier proceso del *CLI*. Esencialmente, es como un almacén de ensamblados compartidos. Existen normas estrictas para su uso y que aseguran un comportamiento correcto. Si el *GAC* se implementa como un subdirectorio concreto dentro de la máquina, cada ensamblado se mueve a dicho subdirectorio desplegando una estructura de más subdirectorios.

Al ser el *GAC* un recurso común, es conveniente usar una herramienta específica para situar ensamblados en el mismo, ya que el mecanismo de despliegue puede variar de una plataforma a otra. Para mover ensamblados al *GAC,* en *SSCLI* se usa la herramienta *gacutil*, que permite instalar, quitar o consultar los diferentes ensamblados que residen en el mismo*,* así como contar las referencias que se hacen a los mismos para evitar desinstalarlos cuando aún están en uso. Esta herramienta se encarga de todos los detalles concretos para mover el ensamblado de la plataforma, evitando al programador conocerlos y haciendo más fácil esta operación.

18.2.3.5 MANIPULACIÓN Y CONFIGURACIÓN DEL PROCESO DE CARGA DE ENSAMBLADOS

Además de la capacidad para manejar diferentes versiones de un ensamblado, los desarrolladores pueden modificar la política usada cuando se van a vincular. Esta política puede ser especificada para cada aplicación, para cada ensamblado y para cada máquina. La utilidad de modificar cómo son cargados los ensamblados reside precisamente en la capacidad de poder lidiar con los problemas que puede ocasionar las nuevas versiones del mismo. De esta forma un *software* puede escoger si usar la nueva versión de un determinado ensamblado o volver a la anterior en caso de que la nueva tenga errores, sea incompatible o carezca de alguna característica.

Cuando los componentes son cargados a memoria, el proceso de carga tiene en cuenta la información del entorno y también la proporcionada por el programador y el administrador del sistema. Dado que el estándar *CLI* no especifica el mecanismo de carga, cada implementación puede usar uno propio. Por ejemplo, el *SSCLI* carga un ensamblado sólo si la versión del entorno de ejecución que contiene la cabecera de metadatos del ejecutable (puesta por el compilador) coincide con la del entorno de ejecución que lo está cargando. Para variar la forma en la que un ensamblado se carga, se usa un fichero de configuración *XML* que le indica al entorno de ejecución las operaciones concretas a realizar.

18.2.3.6 SEGURIDAD EN *ASSEMBLIES*

SSCLI soporta *CAS* (*Code Access Security*), implementación de un mecanismo de seguridad orientado a componentes que incorpora conceptos de seguridad tradicionales de sistemas operativos. Su objetivo es permitir que el código procedente de diferentes fuentes pueda ser combinado en aplicaciones de forma segura. Los programas se ejecutan bajo el control del motor de ejecución, y el código de los componentes es verificado cuando se compila mediante técnicas *JIT*, pero el motor de ejecución puede intervenir si cree que el componente cargado vulnera alguna norma de seguridad.

CAS combina los llamados permisos (*permission*) con evidencias (*evidence*) y políticas (*policy*). Hay dos partes de *CAS*: la fase de carga de ensamblados y la fase de chequeo en tiempo de ejecución.

Los permisos representan capacidades específicas, como la capacidad de leer un archivo. Se usan en concesiones de permisos (*permission grants*) y demandas de permisos (*permission demands*), que son acciones en tiempo de ejecución controladas por el *CAS*. Una concesión de permisos es una autorización basada en una determinada combinación de políticas y evidencias, mientras que una demanda es la verificación de la concesión correspondiente.

Dentro de un ensamblado, los permisos pueden estar asociados con recursos, identidad de código o identidad de usuarios, y son concedidos al código por cada ensamblado en vez de por cada usuario o por cada proceso. Los permisos son aplicados al código bien de forma declarativa (existen atributos personalizados que especifican el comportamiento de acuerdo con la política) o bien imperativa, en cuyo caso el código se escribe para manipular los servicios del *CAS* directamente. *SSCLI* tiene numerosos permisos incorporados y da soporte para la identidad del código (basándose en los *strongnames*), identidades de usuario y autorización. El programador responsable de un ensamblado es el encargado de proporcionar la información con la que el *CAS* debe trabajar. En el código del ensamblado, bien como atributos o como llamadas al *API*, los requisitos de seguridad se especifican a través de las concesiones y demandas de permisos antes mencionadas. Encima de éstas, el usuario, o administrador responsable de que el entorno de ejecución opere de forma correcta, debe especificar entonces la política de seguridad a seguir, a través de un fichero *XML.*

Las evidencias son información acerca de los ensamblados, que se carga y es usada por el *CLI* en unión con las políticas, para tomar decisiones de enlazado acerca de qué permisos se deben conceder y cuales se deben denegar. La evidencia es una información que implícitamente se considera fiable. El entorno de ejecución soporta ciertos tipos de evidencias, como firmas digitales o lugares concretos desde los cuales se pueden cargar ensamblados. Los ensamblados también pueden aportar evidencias adicionales en la forma de peticiones de conjuntos de permisos (*permission set request*).

El conjunto de concesiones de un ensamblado se obtiene combinando evidencias, las demandas del ensamblados y la política en tiempo de ejecución. Para que toda la infraestructura sea segura, la secuencia de carga tiene que ser tratada muy cuidadosamente. Las evidencias que se encuentran junto con el código pueden ser extensibles o personalizadas, y estas evidencias personalizadas sólo se tendrán en cuenta si la política de seguridad que aplica el motor de ejecución las busca durante el proceso de carga. Los ensamblados pueden contener evidencias personalizadas bien directamente, como datos serializados, o bien por programa.

18.2.4 Dominios de Aplicación

Los dominios de aplicación (*application domains*) son una parte imprescindible de la carga de ensamblados en el motor de ejecución. Son elementos arquitectónicos responsables de cargar y descargar ensamblados dentro del motor de ejecución y de proporcionar, mientras están en memoria, aislamiento entre ellos. Aunque el concepto de aislamiento que proveen estos dominios de aplicación es similar al que los sistemas operativos poseen para las aplicaciones dentro de sus espacios de memoria, en realidad todos los dominios de aplicación coexisten en el mismo espacio de direcciones, de forma que pueden compartir servicios como el recolector de basura, etc. Estos dominios poseen mecanismos para externalizar referencias a sus componentes, es decir, que sus componentes pueden establecer canales de comunicación entre diferentes dominios bajo el control del programador. Esto requiere que los hilos de ejecución puedan acceder fuera de los límites de un dominio, y que el motor de ejecución necesite vigilar cuidadosamente estos accesos para mantener el aislamiento.

Los ensamblados siempre se cargan dentro del contexto de un dominio de aplicación. Toda la comunicación desde o hacia procesos externos o componentes en otros dominios es supervisada por el dominio del componente que se comunica. El motor de ejecución posee mecanismos de *remoting* y de *marshaling* que garantizan el aislamiento entre componentes bajo esa supervisión. Si estos mecanismos son demasiado costosos para una aplicación determinada, los procesos tienen el mecanismo alternativo de usar un dominio de aplicación especialmente reservado para ensamblados que compartan información, aunque con el coste de que no se garantiza el aislamiento y la protección entre los ensamblados.

Hay tres dominios estándar dentro de cualquier proceso *SSCLI*. El primero es el dominio del sistema (*system domain*), que es esencialmente un gestor de arranque para aquellos tipos que son parte del proceso de carga del sistema, como `System.AppDomain` y `System.Exception`. El dominio del sistema carga y mantiene un solo ensamblado, llamado *mscorlib*. Este dominio es la base de las búsquedas para ensamblados adicionales. Para tipos que no son del sistema y necesitan ser compartidos, existe un segundo dominio especial llamado dominio compartido (*shared domain*). Los ensamblados que se cargan en este dominio se denominan neutrales en cuanto a dominio, y sus tipos están disponibles directamente para cada dominio del proceso. Para poder cargar un ensamblado en un dominio, éste debe tener un *strongname* y ser fiable. Éste es el dominio al que anteriormente se hacia mención para poder cargar ensamblados sin la penalización de los mecanismos de *marshaling* y *remoting*. Por último, para tipos "normales" como aquéllos desarrollados por un programador en una aplicación estándar, existe el llamado dominio por defecto (*default domain*). Los programadores pueden elegir crear dominios adicionales por programa si no desean usar este último o quieren aislar el código de la aplicación en uno nuevo. En caso de hacer esto, cualquier tipo que se cargue en más de un dominio estará replicado; cada dominio contendrá una representación independiente de las estructuras del tipo.

Una de las características más importantes de los dominios de aplicación es que son la única forma de descargar tipos del motor de ejecución. Cuando un dominio es descargado de memoria, libera todos los recursos que ocupaba antes de liberarse completamente. Un dominio anota todas las instancias y recursos que usa, tanto si proceden de código *managed* o no.

Por último, se debe mencionar la existencia de los llamados componentes ágiles (*agile components*), que son casos especiales en que es permisible y deseable que el estado de un objeto pueda ser usado fuera de los dominios de aplicación. Ejemplos de estos componentes son:

- Cadenas: Estos elementos comunes son inmutables una vez cargados. Se permite copiar y cachear sus contenidos fuera de los dominios para mejorar el rendimiento.
- Objetos de seguridad: Éstos son parte de la infraestructura del motor de ejecución y necesitan la información de estado global del propio motor. Como debe acceder al estado global desde cualquier dominio, es necesario que sean ágiles.
- Tablas de localización: Estas tablas son muy grandes y duplicarlas para cada dominio puede ser demasiado costoso.
- Componentes que son parte de la infraestructura de *remoting*: Estos componentes deben cruzar los límites de los dominios, dado el servicio que implementan.

Este conjunto de componentes ágiles se suele cargar como parte del dominio del sistema, ya que así pueden estar disponibles en todos los contextos.

18.2.5 Eventos

Los eventos son herramientas que los componentes diseñados en lenguajes destinados al entorno *CLI* pueden usar para notificar información a otros elementos que se registren como clientes de dicho componente. Los eventos [MSDNEvents06] son típicamente usados para gestionar la interfaz gráfica de una aplicación, de manera que registren y notifiquen adecuadamente las acciones de un usuario en dicha interfaz, aunque no se debe restringir únicamente su uso a este ámbito. El *CLI* soporta la creación y notificación de eventos, aunque la forma de crearlos y manipularlos puede variar en función del lenguaje usado. A modo de ejemplo, se presenta aquí el proceso de creación y gestión de un evento en el lenguaje *C#*:

- **Declarar un evento**: Para declarar un evento dentro del cuerpo de una clase, se debe declarar primero un tipo delegado para dicho evento:

```
public delegate void ManejadorDeEvento(object sender, EventArgs e);
```

El tipo delegado define una serie de argumentos que son pasados al método que finalmente tratará el evento. Si el programador así lo desea, varios eventos pueden compartir el mismo tipo delegado, por lo que crear un tipo delegado se debe hacer siempre que no interese emplear alguno ya existente. Una vez hecho esto, se debe declarar el evento en sí:

```
public event ManejadorDeEvento cambiado;
```

Los eventos se declaran como si fuesen un atributo cuyo tipo fuese el tipo delegado declarado anteriormente, aunque en este caso se emplea la palabra reservada `event` tal y como se ven en el ejemplo.

- **Invocar un evento**: Una vez que una clase ha declarado un evento, puede tratarlo como si fuese un atributo normal. Ese atributo puede ser `null` (si no hay ningún cliente interesado en el evento) o puede referenciar a un delegado, que debe ser llamado cuando el evento se produce. Por tanto, a la hora de procesar un

evento primero se debe comprobar que el delegado para dicho evento existe, y en caso de que exista, invocarlo. La invocación de un evento sólo puede ser realizada dentro de la clase que lo ha declarado.

```
if (cambiado != null)
        cambiado(this, e);
```

- **Registrarse a un evento**: Desde fuera de la clase que lo declara, un evento es casi idéntico a un atributo, aunque el acceso al mismo está muy restringido. Las únicas operaciones posibles son:

    - Crear un nuevo delegado para ese atributo.
    - Quitar un delegado al atributo.

Estas operaciones se hacen con los operadores `+=` y `-=`. Para recibir invocaciones de eventos, el código de un cliente primero crea un delegado del tipo de evento que se refiere al método que debe ser invocado desde el mismo. Posteriormente, añade ese delegado al resto de delegados que pueden estar conectados a dicho evento, usando `+=`. La operación contraria es también posible:

```
// Se añade "ClaseCambiada" al evento "cambiado" de la clase "Clase":
Clase.cambiado += new ChangedEventHandler(ClaseCambiada);

/*Cuando el código del cliente ya no desea recibir más invocaciones de eventos, se quita el
delegado del evento usando el operador -=*/
Clase.cambiado -= new ChangedEventHandler(ClaseCambiada);
```

Los eventos también pueden ser utilizados con *interfaces*. Existen más consideraciones que se deben tener en cuenta a la hora de trabajar con eventos, pero no se contemplarán en esta sección para no dar un nivel de detalle excesivo y complicar innecesariamente la descripción.

18.2.6 Los Componentes en Detalle

En este apartado se va a describir cómo se "fabrican" componentes en tiempo de ejecución a partir de los datos disponibles para el mismo, creando una estructura que optimice y adapte el componente lo más posible de cara a la ejecución de los programas que los usen. Este proceso transforma su representación, basada en los metadatos residentes en un ensamblado y el *CIL* que representa su comportamiento, en una estructura física e instrucciones propias de un procesador y arquitectura concretos. El motor de ejecución debe ocuparse de dar este "salto" entre ambas representaciones, de manera que el *CIL* se transforme en operandos y *opcodes* mientras que los metadatos se transformarán en estructuras de datos en memoria, que encajen con las restricciones del procesador sobre el que se esté ejecutando la aplicación y las particularidades del sistema operativo. Un enfoque tradicional del proceso de compilación haría que el *frontend* de un compilador procesase la descripción de los tipos de alto nivel y la convierta en una representación intermedia. El *backend* entonces convertiría el *CIL* a un grafo de flujo, lo optimizaría, y generaría código nativo con dos conjuntos de símbolos:

- *Imports*: Que pueden ser usados para localizar direcciones durante el enlazado.
- *Exports*: Cuyas direcciones pueden ser usadas por otros módulos.

En tiempo de enlazado, múltiples módulos se combinarían en una imagen ejecutable única. Las direcciones y *offset* contenidas en su código serían recalculadas si es necesario, sustituyendo los nombres simbólicos por dichas direcciones recalculadas. Una vez producida la imagen ejecutable, el cargador es el responsable de moverla a un espacio en memoria virtual con permisos de ejecución, así como hacer el resto de funciones de recolocación en memoria, cargando por ejemplo librerías *DLL*.

Como se puede suponer por lo ya descrito del *CLI*, esta aproximación tradicional no es aplicable a este entorno de ejecución, ya que si en el momento en el que el código es generado las estructuras son fijadas en memoria y los nombres se traducen a direcciones, la ejecución se hace dependiente de detalles concretos de un procesador específico y entonces el *CLI* no podría ofrecer todas las características para las que fue diseñado. El diseño del *CLI* necesita separar lo más posible el proceso de compilación, enlazado y carga, redistribuyendo además sus tareas: El compilador producirá como salida el lenguaje intermedio *CIL*, mientras que el motor de ejecución será el responsable del resto de tareas. Además, a diferencia del proceso anterior, todos los datos disponibles acerca de la estructura y comportamiento del *software* no son descartados a la hora de construir el ejecutable final, sino que se hacen disponibles para su uso en tiempo de ejecución. Son estos metadatos los que, junto con la estructura en memoria de los componentes, van a ser descritos a continuación.

18.2.6.1 ANATOMÍA DE UN COMPONENTE

En esta sección se describirá como *SSCLI* mantiene una instancia y la información relativa a su tipo en memoria. Estas dos estructuras físicas son bastante complejas, y los elementos que las componen están distribuidos a lo largo de diferentes regiones de memoria. La figura 18.3 presenta un esquema de como se distribuye dicha estructura. Para describir todo este conjunto de información, se partirá de la estructura de una instancia, examinando sus elementos, para describir posteriormente la estructura de un tipo y su creación.

ESTRUCTURA DE UNA INSTANCIA

Aunque en un lenguaje de alto nivel las instancias parecen unidades muy cohesionadas, accesibles completamente a partir de una única referencia, a bajo nivel estas estructuras no son una información monolítica, contenida en una sección de memoria única y contigua del *SSCLI*. Tal y como se ve en la figura 18.3, internamente la información de un objeto se reparte en varios elementos aunque, para poder tener esa imagen de un objeto de alto nivel como una sola entidad cohesionada, siempre se podrá acceder a cada una de las diferentes partes internas del objeto y los datos que contienen a partir de una única referencia al mismo.

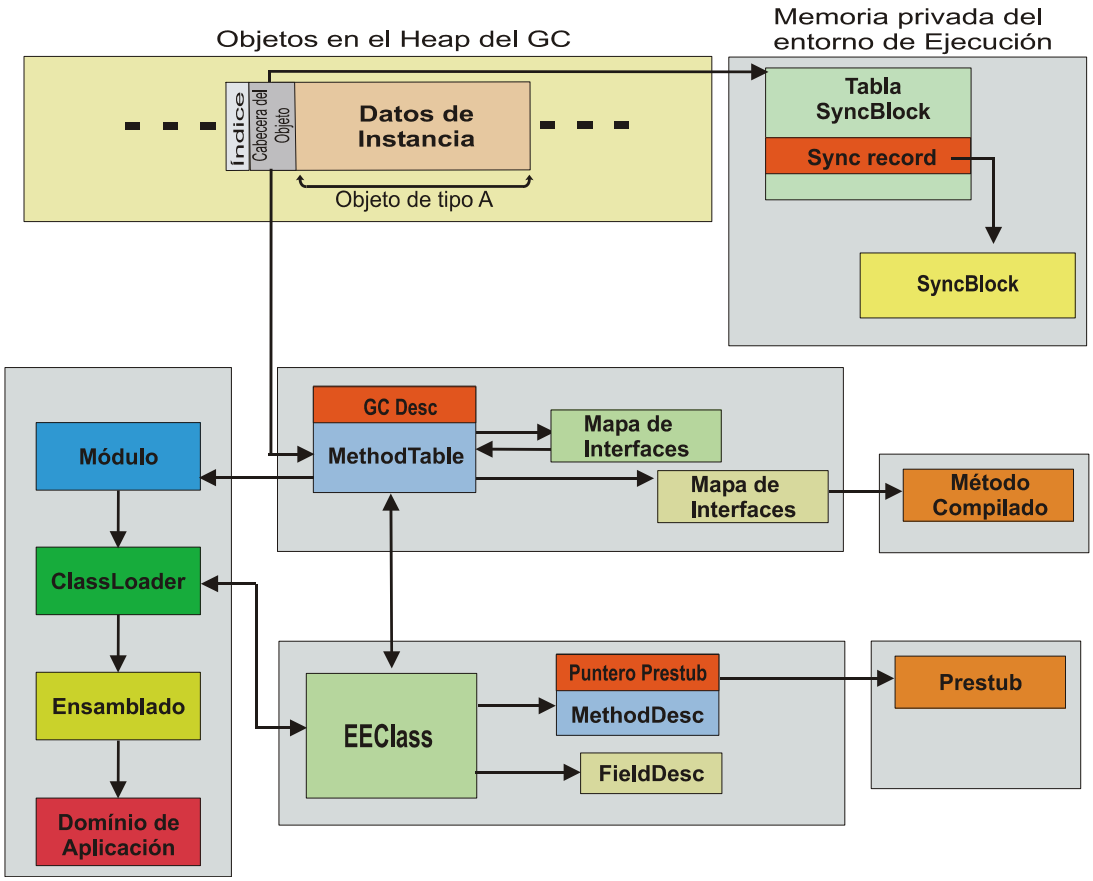

Figura 18.3: Estructura de un componente en *SSCLI* [Stutz03]

A nivel de *CIL*, las instancias son creadas dentro del *heap* del recolector de basura mediante instrucciones como `newobj` y `newarr`. Cada instancia creada de esta forma contiene una referencia a una tabla de punteros a funciones llamada *MethodTable*, tabla cuyas entradas apuntan de forma directa o indirecta al código de sus métodos (su implementación), algo que se verá en el apartado de compilación *JIT*. La tabla de métodos contiene pues datos propios de un tipo, con lo que puede ser compartida por todas las instancias de un tipo dado. Por tanto, todas las instancias de un tipo dado contienen una referencia a una misma *MethodTable*.

Cuando se tiene una referencia a un componente, realmente lo que se tiene es un puntero a su tabla de métodos (*MethodTable*) antes descrita. Mientras que los datos de una instancia residen en el *heap* del recolector de basura o en la pila de un hilo, toda la información usada para la descripción de un tipo, el código compilado, y el contexto del motor de ejecución asociado a los datos de una instancia reside en zonas de memoria privada perteneciente al motor de ejecución. No obstante, toda la información antes descrita puede ser accedida usando el puntero a la tabla de métodos (figura 18.3)

Por tanto, vemos claramente como a bajo nivel no se puede esperar encontrar una clase `Object` perfectamente delimitada con toda la información, sino que la estructura se complica. Esta clase `Object` de bajo nivel aparece como sigue:

```
/*
 * Object
 *
 * This is the underlying base on which objects are built.   The MethodTable
 * pointer and the sync block index live here.  The sync block index is actually
 * at a negative offset to the instance.  See syncblk.h for details.
 *
 */
```

```
class Object
{
  protected:
    MethodTable    *m_pMethTab;

  protected:
    Object() { };
   ~Object() { };

  public:
    VOID SetMethodTable(MethodTable *pMT)
    {
        m_pMethTab = pMT;
    }


    MethodTable    *GetMethodTable() const
    {
        return m_pMethTab ;
    }

//La clase Object es mucho más extensa.
};
```

Aunque puede parecer que un objeto sólo es un puntero a la tabla de métodos, hemos visto como realmente cada instancia tiene asociados más elementos que permanecen "invisibles", usados para que el entorno de ejecución pueda controlar mejor la instancia, por ejemplo en cuanto a gestión en memoria y sincronización.

Una instancia, una vez creada, puede usar varios servicios del motor de ejecución que requerirán la creación y reserva de memoria para las diferentes estructuras de datos necesarias para cada servicio asociado con la misma, como los servicios de manejo de hilos y sincronización. Estas estructuras se encuentran en memoria privada del entorno de ejecución en vez de en el *heap* del recolector de basura, estando así separadas la información de la instancia y la de las estructuras de control. Para acceder desde la instancia a estas estructuras de control se usa el llamado índice de bloque de sincronización o *syncblock index*[13], información que, al igual que el puntero a la tabla de métodos, toda instancia tiene asociada. Sin embargo, acceder a esta estructura de datos no es algo sencillo ni inmediato. La implementación interna de *SSCLI* especifica que toda instancia tiene un *slot* anónimo antes del puntero a la tabla de métodos, y que este *slot* puede ser convertido de forma segura a un objeto de tipo `ObjHeader`, como se ve en el código que implementa la obtención de esta estructura para la clase `Object`:

```
ObjHeader *GetHeader()
{
       return ((ObjHeader *)this) - 1;
}
```

Este mecanismo permite a cada instancia guardar información que no es formalmente parte de la clase `Object`, de forma que una vez que se tiene la información de una instancia, se puede saber rápidamente dónde está la información relativa a la administración. La forma vista de estructurar las instancias asegura la máxima eficiencia en ejecución, uno de los principales objetivos de la construcción del *SSCLI*. La definición de la clase *ObjHeader* en *SSCLI* es la siguiente:

```
class ObjHeader
{
  private:
  // !!! Notice: m_SyncBlockValue *MUST* be the last field in ObjHeader.
```

13 *Realmente puede contener información que no sea relativa a tareas de sincronización, aunque el nombre pueda llevar a engaño.*

```
  DWORD  m_SyncBlockValue;      // the Index and the Bits

  public:

    // Access to the Sync Block Index, by masking the Value.
    DWORD GetHeaderSyncBlockIndex()
    {

        // pull the value out before checking it to avoid race condition
        DWORD value = m_SyncBlockValue;
        if (value & BIT_SBLK_IS_APPDOMAIN_IDX)
            return 0;
        return value & MASK_SYNCBLOCKINDEX;
    }
//El código de la clase ObjHeader sigue, es mucho más extenso del mostrado
};
```

Como puede verse en la figura 18.3, aunque tanto `Object` como `ObjectHeader` son clases distintas, en memoria son situadas siempre de forma contigua, de forma que la cabecera siempre se podrá encontrar antes de los datos de una instancia. En el código de la clase cabecera podemos encontrar el atributo `m_syncBlockValue`, que puede contener diversos valores: Un valor `LONG` que se pueda usar como índice o un campo de *bits* usado como *flags* para diferentes propósitos. Si el valor es un índice y no es 0, entonces el índice será un desplazamiento dentro de la tabla global `g_pSyncTable` donde se podrá encontrar el *syncblock* propiamente dicho, con su información asociada.

INFORMACIÓN ACERCA DE LA ESTRUCTURA: LOS METADATOS

Una vez vista la estructura de los datos contenidos por una instancia, se describirá a continuación la organización de la información disponible en tiempo de ejecución de un tipo. Esta información puede ser solicitada por un programador usando métodos de la clase `System.Object` y es también necesaria para el funcionamiento correcto del entorno de ejecución durante los procesos de compilación, recolección de basura, resolución de llamadas virtuales y otros servicios.

Como ya se ha visto, la carga de ensamblados en *SSCLI* es el primer paso del proceso de convertir descripciones de tipos de su formato original (diseñado específicamente para ser un soporte eficiente para la persistencia de la información) a estructuras en memoria y secuencias de *opcodes*. Una vez cargado un ensamblado de esta forma, los metadatos se convierten a un formato diferente basado en punteros, que puede ser combinado fácilmente con la información del motor de ejecución para su carga eficiente en memoria. El formato en que estos datos se cargan en memoria divide los mismos en dos estructuras diferentes, una que contiene una serie de datos que deben ser accedidos de forma rápida durante la ejecución de un programa, y otra que contiene datos estructurales que típicamente son necesarios sólo por compiladores o el *API* de reflexión. En la figura anterior, la clase `MethodTable` contiene todos los datos de rápido acceso, mientras que la clase `EEClass` es la clase que contiene el otro tipo de datos.

Tablas de Métodos:

La tabla de métodos es una estructura compleja que consiste en una cabecera seguida de una tabla de longitud variable de punteros a métodos e interfaces. También posee una clase adicional llamada *GCDesc*, usada para labores de recolección de basura y situada antes de la cabecera, en un *offset* negativo, de forma contigua a la tabla de métodos. La definición de la clase *MethodTable* puede verse a continuación:

```
class MethodTable
{
public:

        DWORD           m_wFlags;               // Low DWORD is component size for array
                                                // objects or value classes, zero otherwise

        DWORD           m_BaseSize;             // Base size of instance of this class
        EEClass*        m_pEEClass;             // class object

        union
        {
            LPVOID*     m_pInterfaceVTableMap;  // pointer to subtable for interface / vtable
                                                // mapping
            GuidInfo*   m_pGuidInfo;            // The cached guid inforation for interfaces.
        };

    // };


    WORD                m_wNumInterface;        // number of interfaces in the interface map
    // The CorElementType for this class ( most classes = ELEMENT_TYPE_CLASS)
    BYTE                m_NormType;
    Module*             m_pModule;

    WORD                m_wCCtorSlot;           // slot of class constructor

    // slot of default constructor
    WORD                m_wDefaultCtorSlot;
    InterfaceInfo_t*    m_pIMap;                // pointer interface map for classes.

protected:

    // for interfaces, this is the default stub to give out
    // this could be a specialized stub if there is only
    // one introduction of this interface

    // total slots in this vtable
    DWORD   m_cbSlots;

public:

    // vtable slots follow - variable length
    // Unfortunately, this must be public so I can easily access it from inline ASM.
    SLOT    m_Vtable[1];

//El código de esta clase es mucho mayor que el mostrado aqui.
};
```

Como puede verse, la información que posee una tabla de métodos es muy importante y frecuentemente accedida, por lo que se han tomado medidas especiales para asegurar un rendimiento lo más alto posible. Contiene uniones para cierta información que puede estar solapada, y ahorrar memoria al máximo, así como una optimización especial para acceder al *array* de punteros a métodos. Esta clase contiene sólo un *array* de una entrada como tabla de métodos (*SLOTS*), pero esto no limita la tabla de métodos a un solo elemento, sino que se hace para evitar una indirección extra a la hora de acceder a ellos. Cuando se carga esta clase en memoria, el entorno de ejecución carga la tabla de métodos justo a continuación de la clase *MethodTable* (ver el siguiente dibujo), de manera que para acceder a uno en concreto, una vez que se tiene la dirección de comienzo y el número de métodos existente, se pueda acceder directamente, sin tener que reservar memoria dinámica para esta tabla ni usar una indirección extra a un puntero para su acceso. Nótese también como la tabla de métodos posee un puntero a la *EEClass* para acceder al resto de información.

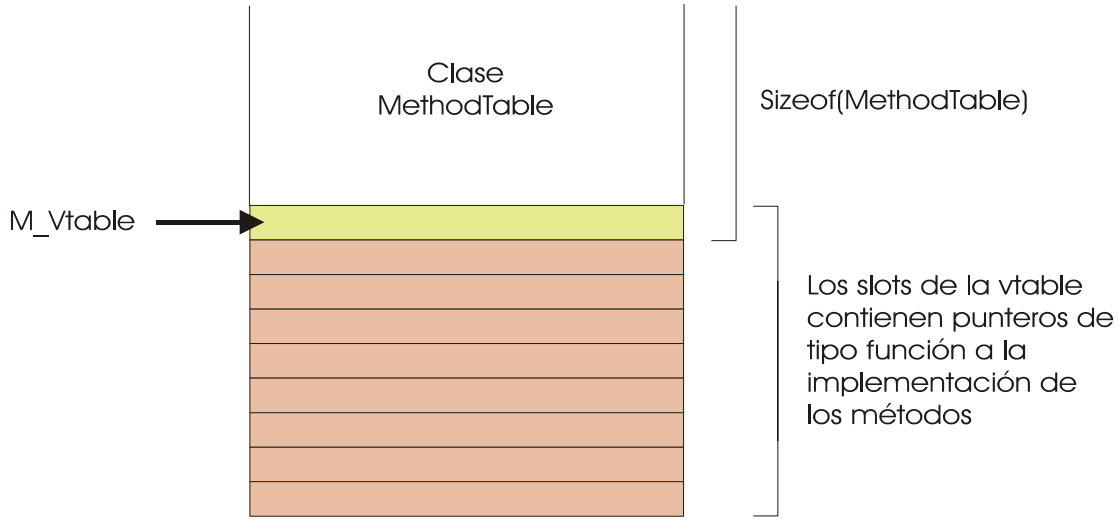

Figura 18.4: Estructura de una *MethodTable* en memoria [Stutz03]

El orden de los *slots* dentro de la clase no es tampoco aleatorio. Cada *slot* contiene un puntero a una pieza de código ejecutable de una función que sigue el convenio de llamada del *CIL*, pero se pueden encontrar tres secciones dentro del *array*, que contienen métodos en el siguiente orden del primero al último:

- Los métodos estáticos para el tipo descrito.
- Los métodos heredados por el tipo.
- Los métodos introducidos, es decir, los que el tipo define explícitamente.

Mezclados entre alguna o todas las secciones descritas en el *array* también se puede encontrar el acceso a la implementación de los métodos que el tipo tiene por las interfaces que implementa. Para una descripción más precisa de cómo se integran los interfaces dentro de la tabla de métodos se recomienda consultar [Stutz03].

EEClasses:

La definición de la clase *EEClass* especifica una serie de atributos que comienzan por `bmt`, abreviatura de *BuildMethodTable*, y que son usados para construir la tabla de métodos antes descrita. Además de esto, posee los siguientes atributos:

```
class EEClass {


protected:

    EEClass *m_pParentClass;
    WORD   m_wNumVtableSlots;  // Includes only vtable methods (which come first in the table)
    // Includes vtable + non-vtable methods, but NOT duplicate interface methods
    WORD   m_wNumMethodSlots;
    // value classes have some duplicate slots at the end
    WORD   m_wDupSlots;

    WORD   m_wNumInterfaces;


    // We have the parent pointer above.  In order to efficiently backpatch, we need
    // to find all the children of the current type.  This is achieved with a chain of
```

```
    // children.  The SiblingsChain is used as the linkage of that chain.
    //
    // Strictly speaking, we could remove m_pParentClass and put it at the end of the
    // sibling chain.  But the perf would really suffer for casting, so we burn the space.
        EEClass *m_SiblingsChain;

        union
        {
                // coclass for an interface
                EEClass* m_pCoClassForIntf;
                // children chain, refer above
                EEClass *m_ChildrenChain;
        };


private:
    // Number of fields in the class, including inherited fields
    WORD   m_wNumInstanceFields;
    WORD   m_wNumStaticFields;

    // Number of pointer series
    WORD    m_wNumGCPointerSeries;

    // Number of static handles allocated
    WORD    m_wNumHandleStatics;

    // # of bytes of instance fields stored in GC object
    DWORD   m_dwNumInstanceFieldBytes;
    ClassLoader *m_pLoader;

    // includes all methods in the vtable
    MethodTable *m_pMethodTable;

    // a pointer to a list of FieldDescs declared in this class
    // There are (m_wNumInstanceFields - m_pParentClass->m_wNumInstanceFields +
    // m_wNumStaticFields) entries in this array
    FieldDesc *m_pFieldDescList;

    // Number of elements in pInterfaces or pBuildingInterfaceList
    DWORD   m_dwAttrClass;
    DWORD   m_VMFlags;

    BYTE    m_MethodHash[METHOD_HASH_BYTES];

    SecurityProperties m_SecProps ;

    mdTypeDef m_cl; // CL is valid only in the context of the module (and its scope)


    MethodDescChunk     *m_pChunks;

    WORD    m_wThreadStaticOffset;  // Offset which points to the TLS storage
    WORD    m_wContextStaticOffset; // Offset which points to the CLS storage
    WORD    m_wThreadStaticsSize;   // Size of TLS fields
    // Size of CLS fields
    WORD    m_wContextStaticsSize;

    static MetaSig      *s_cctorSig;
}
```

Podemos ver como existen una serie de atributos que contienen información estructural, como por ejemplo los atributos que pertenecen al tipo. La lista de atributos es un conjunto de clases *FieldDesc* que guardan información completa acerca de cada uno de los atributos definidos por el tipo. No obstante, si se quieren consultar los atributos heredados se debe acceder a los atributos de las clases padre, dato también disponible en la definición de la *EEClass*. La clase *FieldDesc* es la siguiente:

```
class FieldDesc
{
protected:
    // Note, 2 bits of info are stolen from this pointer
    MethodTable *m_pMTOfEnclosingClass;
```

```
    // Strike needs to be able to determine the offset of certain bitfields.
    // Bitfields can't be used with /offsetof/.
    // Thus, the union/structure combination is used to determine where the
    // bitfield begins, without adding any additional space overhead.
    union {
            struct {
             unsigned char m_mb_begin;
                } offset1;
        struct {
            unsigned m_mb               : 24;

            // 8 bits...
            unsigned m_isStatic         : 1;
            unsigned m_isThreadLocal    : 1;
            unsigned m_isContextLocal   : 1;
            unsigned m_isRVA            : 1;
            unsigned m_prot             : 3;
            unsigned m_isDangerousAppDomainAgileField : 1; // This is used in checked only
        } u;
    };

    // Strike needs to be able to determine the offset of certain bitfields.
    // Bitfields can't be used with /offsetof/.
    // Thus, the union/structure combination is used to determine where the
    // bitfield begins, without adding any additional space overhead.
    union {
            struct {
             unsigned char m_dwOffset_begin;
                } offset2;
        struct {
            // Note: this has been as low as 22 bits in the past & seemed to be OK.
            // we can steal some more bits here if we need them.
            unsigned m_dwOffset         : 27;
            unsigned m_type             : 5;
        } v;
    };

//El código de esta clase es más extenso que el aqui mostrado.
};
```

Para mejorar la eficiencia, la clase *FieldDesc* se ha hecho lo más compacta posible. Para ello, como se puede apreciar en el código de arriba, se usan campos de *bits* y uniones que ahorren memoria. En cuanto a los métodos, la información de sus metadatos también se guarda en la *EEClass* usando un conjunto de estructuras *MethodDescChunk*. Cada una de esas entradas es un conjunto de estructuras *MethodDesc* puestas juntas para un acceso rápido. Al igual que los atributos, para encontrar la información de los métodos introducidos por las superclases es necesario recorrer los métodos de las mismas. *MethodDesc* es una estructura compacta, que tiene pocos datos y los métodos para acceder a ellos y manipularlos, así como otros elementos que permiten la ejecución del código asociado al mismo. Todo *MethodDesc* tiene además un atributo adicional oculto en un *offset* negativo llamado *thunk*. Este atributo se describirá en la sección siguiente ya que forma parte del proceso de compilación.

En resumen, en *SSCLI* las instancias siempre contienen un puntero a una *MethodTable*, que a su vez referencia a una *EEClass*. Aunque estas estructuras están separadas para mejorar la eficiencia, ambas son partes de una misma estructura lógica y definen las propiedades estructurales de una clase en tiempo de ejecución, siendo inicializadas a partir de los metadatos existentes en los ensamblados.

18.2.7 Compilación JIT en SSCLI

18.2.7.1 PROCESO DE COMPILACIÓN GENERAL

Una vez que la *EEClass* y su *MethodTable* relacionada han sido completamente configuradas, estarán listas tanto la información necesaria para la compilación como la mayoría de las estructuras necesarias para una ejecución correcta. En este punto, el motor de ejecución podrá compilar y ejecutar el código contenido en el tipo, pero dado que este proceso es retrasado hasta el último momento posible, se debe estudiar que es lo que desencadena la compilación *JIT* en este sistema.

En lenguajes tradicionales, como *C++*, la compilación de un método concreto ocurre en cuanto sea posible, independientemente de que este método se use o no. Un compilador de *C++* necesita eliminar la mayor cantidad de información posible acerca de la estructura del programa en el momento de la ejecución, para minimizar la cantidad de procesamiento necesario y optimizar así al máximo la ejecución del programa. No obstante, esto causa que existan métodos compilados pero nunca se usen, provocando así una utilización de recursos no eficiente.

Para evitar esta situación y poder lanzar la compilación en el último momento que sea posible, de manera que sólo se compile aquel código que realmente vaya a ser utilizado, se necesita incorporar un mecanismo que permita informar al motor de ejecución de que la llamada a un método concreto va a tener lugar y que, en caso de no haber sido ya compilado el código de ese método, haga que el compilador *JIT* sea llamado justo antes de intentar comenzar la ejecución del mismo, de forma que se compile sólo una vez: La primera vez que se llama. Este proceso podría hacerse analizando el programa y buscando estáticamente en él todas las invocaciones de métodos que hace, forzando al compilador *JIT* a procesar sólo aquéllos que use dicho programa en su código en ese momento, pero el coste de esta solución seria imposible de asumir con programas grandes.

En vez de eso, el *CLI* emplea una técnica por la cual usa una llamada indirecta a una función ayudante (*helper function*) llamada *prestub helper*. Aunque una *MethodTable* debería contener punteros al código nativo, que se debería ejecutar cuando se haga la llamada a la función correspondiente, de manera que, para que un código sea ejecutado, simplemente se deba localizar el puntero correspondiente al método y acceder a través de él a su código, esto no ocurre exactamente así. Cada uno de los *SLOTS* de la *MethodTable* se cargará inicialmente con un *thunk* (entendiendo éste como una pequeña función ayudante se inserta automáticamente por el entorno de ejecución), que lanzará la compilación *JIT* cuando se llama y hará un proceso de *backpatching* en la tabla cuando esta termine (cambiando el puntero existente por otro que apunte finalmente al código de la función recién compilada). El código que realiza toda esta función se denomina *stubcall*.

El proceso se describe ordenadamente a continuación: Cuando un método aún no compilado se llama, el *stubcall* pone la dirección del *MethodDesc* correspondiente al método llamado en la pila (o en un registro) y entonces ejecuta un salto al *prestub helper*. El *MethodDesc* es necesario durante la compilación dada la información específica del método que contiene, que es necesaria a la hora de generar código. *SSCLI* usa el *stubcall* para guardar la localización del *MethodDesc,* uniendo el código de cada *stubcall* directamente al cuerpo del *MethodDesc* que le corresponda usando unas líneas de código ensamblador. Este código se crea durante la inicialización del *MethodDesc*, usando funciones especificas de cada procesador, dentro de un *offset* negativo respecto al cuerpo principal del mismo, y consiste en una instrucción *call* a la dirección del *prestub helper*, pasada como parámetro en tiempo de ejecución. De esta forma, el *MethodDesc* de un

método cualquiera está siempre relacionado con su *stubcall* y no es necesario hacer búsquedas de ambos elementos, de manera que el *stubcall* pueda situar la dirección del *MethodDesc* en la pila de forma eficiente. El *prestub helper* es pues una función compleja que origina todo el procesamiento antes y después de la llamada y la compilación *JIT*, y su función se puede resumir en los siguientes pasos:

- El motor de seguridad y el de *remoting* pueden intervenir primero para interceptar y chequear la llamada que se está realizando, por si tuvieran que intervenirla.
- El *profiler* incorporado en *SSCLI* puede extraer información o incluso modificar el *CIL* del método si fuese necesario.
- Si es la primera vez que el tipo se usa, se debe comprobar si el constructor de dicho tipo ha sido llamado.
- El *SLOT* para el que el cuerpo del método se crea es sometido a *backpatching*, proceso mediante el cual dicho *SLOT* ya no apuntará al código que origina el mecanismo de compilación visto, sino que lo haría a la dirección de inicio del código que acaba de ser compilado. Esto no lo hace directamente, como veremos a continuación.
- Existen además otros chequeos especializados, como por ejemplo si la llamada es una operación de *unboxing* (en la que se debe generar código específico para esta tarea) o casos similares que deben tenerse en cuenta.

Una vez que el *prestub helper* ha compilado un método, la dirección del cuerpo del método no se inserta directamente en el *SLOT* correspondiente de la clase *MethodTable*, ya que la memoria que contiene puede ser reciclada posteriormente. En vez de eso se crea un segundo *thunk*, contenido en una estructura de tipo *JittedMethodInfo*. Este código es llamado *JMI thunk* y sí apuntará esta vez al cuerpo del método compilado anteriormente. El *JMI thunk* es pues una indirección y su propósito no es otro que causar la recompilación del método en caso de que la memoria que ocupaba haya sido reciclada por el recolector de basura si necesita recursos. El proceso descrito es el *backpatching* completo que se realiza en *SSCLI*, y asegura que los *SLOT* sólo sean modificados bajo demanda. Incluso si hay varios *slots* apuntando al mismo método, todos son modificados de forma independiente cuando son usados, aun compartiendo el mismo *JMI thunk*.

18.2.7.2 VERIFICACIÓN DE CÓDIGO Y COMPILACIÓN *JIT*

Para verificar que el código es seguro en cuanto al uso de sus tipos, el compilador *JIT* debe recorrer el *CIL* para asegurarse de que cada instrucción se comporta de acuerdo con las reglas especificadas en el estándar *ECMA*. Dado que este proceso se produce cuando el *CIL* es transformado en código nativo, la compilación *JIT* y la verificación se mezclan, es decir, ocurren dentro de la misma operación. Para verificar que el *CIL* sea válido para cada método, y que todos los posibles caminos de ejecución del código sean seguros, el estándar *ECMA* [ECMA02] describe un conjunto de tipos de verificación más básicos que los que el propio *CIL* posee, y especifica cuidadosamente todo el proceso de verificación del *CIL* para que las implementaciones del estándar tengan un modelo válido del que partir para hacer esta tarea.

La verificación y la compilación dentro del *SSCLI* se hacen con un algoritmo de una sola pasada. Cuando el compilador *JIT* procesa una sección del *CIL,* también verifica que los operandos son válidos para las operaciones que se llevan a cabo y que la pila no se desbordará cuando el código se ejecute. Para cualquier salto, el *JIT* se asegura también de que los tipos en la pila se traten consistentemente por cada rama del mismo,

usando una estructura interna que guarde los estados de la pila asociados con cada una de ellas.

La verificación del código de un método no se hace recorriendo el método de principio a fin directamente, ya que el *JIT* necesita conocer siempre el estado de la pila. Para evitar aquellas partes que pueden introducir un estado no determinado en la pila, el orden de proceso se cambia, de forma que los lugares en los que hay un estado de la pila indeterminado son marcados para ser comprobados posteriormente, marcas que son situadas en una estructura llamada *split stack,* cuyo tamaño varía en función del número y tipo de ramificaciones encontradas. Cuando la *split stack* está vacía y el conjunto de *opcodes* a procesar se acaba, el método completo se da por verificado y compilado a código nativo satisfactoriamente, eliminando el código muerto que pudiese existir.

El proceso de compilación se inicia por el método *jitCompile* de la clase *Fjit*. Para comenzar correctamente, el *JIT* comprueba que no hay excepciones cuyos bloques se extiendan más allá del final del método o bien tengan tamaño 0, realizando posteriormente el resto de comprobaciones relacionadas con excepciones. Las variables locales son inicializadas a 0 ó a `null` si son referencias. Al comenzar la compilación propiamente dicha, tanto los bloques `try` como sus correspondientes bloques `catch`, `finally`, `fault` o `filter` son colocados en el *split stack*, recordando también el estado de la pila que se espera encontrar en esos puntos, para empezar posteriormente a procesar cada una de las instrucciones *CIL* por el punto de entrada del método. Mientras compila cada *opcode*, el compilador toma nota tanto de los contenidos del *stack* como del estado de las posibles ramificaciones del código para el uso del verificador. En caso de encontrar un salto, el verificador puede actuar de dos formas:

- En caso de ser un salto incondicional, el verificador comprobara si el *offset* de destino ha sido ya jiteado, y si lo ha sido, se asegurará que el estado de la pila encaja con lo que se espera. En caso de no haber sido jiteado, el verificador recordará el estado de la pila y continuará en el *offset* de destino. Si el *split stack* está vacío, la compilación del método se termina, pero en caso contrario se le añade un nuevo *offset* y se continúa la compilación.
- Si el salto es condicional, se procesa el código de forma similar con una variante la técnica descrita.

El proceso de compilación de instrucciones *CIL* se hace procesándolas de una en una, y está construido de forma que pueda portarse más fácilmente. El compilador sitúa las instrucciones nativas ya compiladas en un *buffer* de salida. Los *opcodes* del *CIL* se procesan en una gran estructura `switch,` donde hay una rama para cada una de las posibles instrucciones del lenguaje. Para cada una de esas ramas `case`, finalmente el *CIL* emite una serie de instrucciones nativas en el *buffer* de salida mencionado. El compilador *JIT* que usa *SSCLI* por defecto no introduce optimizaciones al código generado para permitir una generación de código más rápida y una mejor comprensión del mismo.

A la hora de generar código, el compilador *JIT* tiene un conjunto de capas de marcos que son usadas para emitir código. En el núcleo de éstas se encuentran una serie de funciones de ayuda que están diseñadas para ser portadas fácilmente a otras plataformas, además de una serie de macros específicas para ciertas plataformas concretas. Modificando esas macros y funciones de ayuda es posible generar de forma relativamente sencilla código de *SSCLI* para cualquier plataforma a la que se desee portarlo, al tener que portar teóricamente sólo el código de esos elementos.

18.2.7.3 CONVENIOS DE LLAMADA

Una vez verificado y compilado el *CIL*, el código del método puede ser ejecutado de forma segura. Como el *CLI* permite el uso de recursividad, la pila se usa para guardar el estado de ejecución. Cada llamada a un método introduce un registro de activación en esa pila, conteniendo sus argumentos, valor de retorno, variables locales y otra información necesaria para otras tareas como seguridad (el objeto de seguridad usado por el *CAS* (*Code Access Security*)).

Como los métodos llaman a otros métodos, la pila se mantiene cooperativamente usando una serie de convenios de llamada. Todas las llamadas comienzan con el inicialización de un *callsite* (el contexto de la pila asociado con la llamada a un método por el llamante). Los parámetros son siempre parte del *callsite*, ya que sólo el que llama puede introducirlos en él, puesto que el método que se llama no los conoce. No obstante, diferentes convenios de llamada pueden usar otros mecanismos. Se describirá a continuación cómo interoperan estos convenios de llamada dentro del *SSCLI*.

CONVENIO DE LLAMADA DEL *JIT*:

Este convenio de llamada es el que usa de forma estándar en el código que produce el compilador *JIT*. Desde la perspectiva del *CIL*, hay cuatro formas diferentes de llamar a código:

- La instrucción `jmp`, que no es verificable.
- Tres formas diferentes de la instrucción `call`: `calli`, `call` y `callvirt`.

Cada una de esas instrucciones tiene semánticas ligeramente diferentes, y pueden ser modificadas para ser un `tailcall` (optimización que reutiliza el mismo registro de activación durante las llamadas recursivas en vez de generar nuevos registros). La instrucción `call` no es virtual, ejecuta explícitamente el método apuntado por dicha instrucción. La instrucción `callvirt`, en cambio, hace las llamadas indirectamente usando una *vtable*, permitiendo así llamadas a métodos virtuales. Ambas instrucciones reciben un *token* como argumento, sólo cambia el método de búsqueda del mismo: Mientras `callvirt` selecciona una tabla de punteros a métodos en tiempo de ejecución, `call` siempre llama a aquella tabla que es parte del componente sobre el que se llama. Para llamadas indirectas, con la instrucción `calli`, se carga en la pila un puntero a función y se le transfiere el control al mismo.

En el convenio de llamada del *JIT* usado por todas estas instrucciones, tanto los argumentos como el valor de retorno se pasan en la pila, y ninguno de ellos se introduce en un registro (las máquinas abstractas de pila no tienen registros). Los argumentos se introducen inicialmente en el mismo orden que se especifican en el *CLI*, y si la llamada tiene un número variable de argumentos, el *token* "*varargs*" es también introducido antes de ellos, introduciéndose finalmente el valor de retorno automáticamente. Finalmente, si la llamada es un método de instancia, el puntero `this` será el último argumento introducido en la pila y por tanto será también el primero que se encontrará en la misma. Si no hay puntero `this` (método estático), será el *buffer* de retorno el primer parámetro que se encuentre, aunque ambos son opcionales.

CONVENIO DE LLAMADA NATIVO:

El convenio de llamada del *JIT* es simple, pero tiene que lidiar con el convenio de llamada específico del procesador sobre el que finalmente se ejecutará el código. El *SSCLI* debe pues convertir el convenio de llamada del *JIT* al nativo. Para ilustrar este proceso, se presentará a continuación cómo se realiza esta operación en una plataforma *x86*. Cuando el compilador *JIT,* que genera código para una plataforma *x86,* se encuentra con una instrucción `call`, `calli` o `callvirt`, los argumentos ya han sido puestos en la pila siguiendo el convenio ya descrito. El compilador entonces localizará dos de esos argumentos, que puedan caber en cuatro *bytes*, para introducirlos en los registros *ECX* y *EDX*. La pila entonces se compacta para eliminar el espacio dejado por esos argumentos, introduciendo posteriormente *EDX* y luego *ECX* en la pila. Un método como:

```
void f (int32 a, int64 b, int64 c, int32 d, int32 e)
```

Será transformado como indica la siguiente figura:

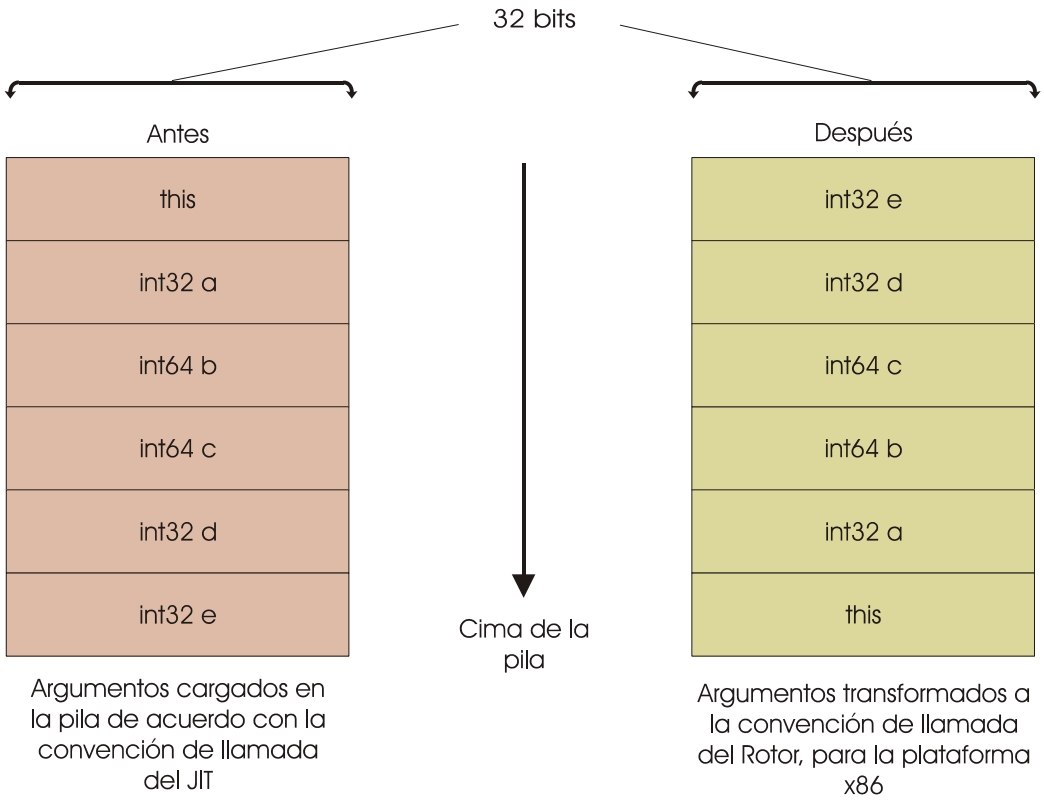

Figura 18.5: Convenciones de llamada en *SSCLI* [Stutz03]

El resto del proceso se ocupará de efectuar la llamada siguiendo el proceso antes descrito, adaptando los convenios a la arquitectura *x86*. Para ello, se usa un conjunto de macros de generación de código, cuyo nombre empieza siempre por *x86*. Para otras arquitecturas también se usa un conjunto de macros equivalente (por ejemplo, los *Power PC* usan las macros que comienzan por *ppc*). El uso de estas macros facilita la portabilidad del sistema, ya que la generación de código puede ser cambiada teóricamente reprogramando sólo estos elementos.

Existen otros convenios de llamada en *SSCLI* usados en código que no es

procesado por el compilador *JIT*, como las funciones de ayuda y los *stubs*, lo que ocasiona que el *stack* no aparezca de una forma homogénea en tiempo de ejecución.

18.2.8 Programación Generativa: Emisión de Código

Para ilustrar mejor las capacidades del *CLI,* y por la especial relación con el trabajo que se ha realizado en esta tesis, ilustraremos ahora cómo se crea código en tiempo de ejecución. Como ya se mencionó anteriormente, el *CLI* posee la capacidad de generar nuevos tipos en tiempo de ejecución, es decir, de emplear la llamada programación generativa. Además de poder examinar la estructura de los metadatos en tiempo de ejecución mediante las funciones del espacio de nombres *System.Reflection*, también posee la habilidad de emitir tipos completamente sintetizados dinámicamente a través del paquete *System.Reflection.Emit*. Aquí sólo se mostrará un ejemplo y una breve descripción de uso; Para un análisis más en profundidad de esta parte del *CLI* se recomienda consultar [Stutz03]. El siguiente ejemplo muestra la síntesis dinámica de un tipo completa para construir el programa "Hola Mundo":

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Threading;

public class EmitHelloWorld
{
      static void Main(string[] args)
      {
            // Creamos un módulo y un ensamblado dinámicamente
            AssemblyName assemblyName = new AssemblyName();

            assemblyName.Name = "HelloWorld";
            AssemblyBuilder assemblyBuilder =
                              Thread.GetDomain().DefineDynamicAssembly(assemblyName,
                              AssemblyBuilderAccess.RunAndSave);

            ModuleBuilder module;

            module = assemblyBuilder.DefineDynamicModule("HelloWorld.exe");

            // Creamos un tipo nuevo para contener el método Main
            TypeBuilder typeBuilder = module.DefineType("HelloWorldType",
                                    TypeAttributes.Public | TypeAttributes.Class);

            // Creamos el metodo Main(string[] args)
            MethodBuilder methodbuilder = typeBuilder.DefineMethod("Main",
                              MethodAttributes.HideBySig | MethodAttributes.Static |
                              MethodAttributes.Public, typeof(void), new Type[]
                              { typeof(string[]) });

            // Generamos el CIL para el metodo Main
            ILGenerator ilGenerator = methodbuilder.GetILGenerator();
            ilGenerator.EmitWriteLine("hello, world");
            ilGenerator.Emit(OpCodes.Ret);

            // Creamos el tipo
            Type helloWorldType = typeBuilder.CreateType();

            // Invocamos el método del tipo
            helloWorldType.GetMethod("Main").Invoke(null, new string[] {null});

            // Fijamos el punto de entrada de la aplicación y la grabamos
            assemblyBuilder.SetEntryPoint(methodbuilder, PEFileKinds.ConsoleApplication);
            assemblyBuilder.Save("HelloWorld.exe");
      }
```

```
}
```

El proceso seguido puede dividirse en siete pasos:

- Creación de un nuevo dominio de aplicación, si fuese necesario, para el tipo. Si se crea un dominio de aplicación nuevo para cargar el tipo nuevo, podrá ser descargado de memoria cuando se desee.
- Para el dominio de aplicación usado, se creará un *AssemblyBuilder* llamando al método *DefineDynamicAssembly*, pasando en la clase *AssemblyName* el nombre para el nuevo ensamblado y proporcionando también las restricciones de acceso.
- Definir cada módulo nuevo para el ensamblado.
- Generar los tipos deseados. Los atributos de cada tipo se describen usando la clase *TypeAttributes* como segundo parámetro en la llamada *DefineType*.
- Definir los miembros de los tipos dinámicos. Para crear métodos, se usa la clase *MethodBuilder* (junto con la signatura del método a definir)*, para* los atributos se usa *FieldBuilder*, para los eventos *EventBuilder*, etc., pasando los atributos deseados según sea necesario.
- Para los métodos, emitir los *opcodes* en *CIL* que formen el cuerpo del mismo.
- Llamar a *CreateType* para finalizar. Con esto el tipo está listo para ser usado e incluso se puede fijar un punto de entrada para ejecutarlo, como en el ejemplo.

Este proceso ilustra cómo la flexibilidad del modelo permite incorporar tipos nuevos a voluntad, creando las estructuras que los mantienen tanto a alto como a bajo nivel de forma dinámica, e integrando esos tipos nuevos en el motor de ejecución junto con los tipos ya existentes (ya que tendrán las misma representación interna), de forma que se usen de la misma manera que aquéllos que se declararon "normalmente".

18.2.9 Hilos

Los hilos en *SSCLI* son, junto con las excepciones, uno de los mecanismos usado para establecer un control de los procesos que hay en ejecución dentro del sistema. Estos dos mecanismos están fuertemente ligados a la pila de ejecución. En *SSCLI* los hilos tienen dos funciones principales:

- Son vehículos para poder ejecutar código, cuya ejecución y tiempo de uso de los recursos del sistema estarán controlados por la implementación existente en el sistema operativo de estos mecanismos, así como las dependencias establecidas entre tareas.
- Son estructuras de datos muy importantes para mantener la información del motor de ejecución acerca del código en ejecución. Estas estructuras mantienen información acerca de la ejecución y la sincronización, incluyendo aspectos de seguridad, de recolección de basura, variables, etc.

Cada hilo de *SSCLI* se implementa sobre un hilo de la *PAL*, por lo que permanecen aislados de los detalles concretos del sistema sobre el que se ejecutan, ya que la *PAL* es

la que se encarga de adaptar el código a los detalles concretos del sistema nativo. Esto es muy importante, ya que la implementación de hilos en las diferentes plataformas puede variar enormemente, y mediante este mecanismo todos los servicios de la plataforma que requieran el uso de hilos verán siempre la misma abstracción. No obstante, por este motivo la implementación de la *PAL* es muy compleja.

Por tanto, dentro de *SSCLI* existen dos tipos de hilos, los hilos de la *PAL* y los hilos *managed*, siendo estos últimos instancias del tipo *Thread*. Los *managed threads* encapsulan un *thread* de la *PAL* y por tanto siempre tienen un *thread* de este último tipo asociado. Sin embargo, un *thread* de la *PAL* no tiene porque tener un *managed thread* asociado necesariamente.

Cada *thread* de la *PAL* puede mantener información privada que represente el estado de su *managed thread* asociado. Además, para permitir la interoperabilidad entre código *managed* y no *managed*, los hilos *managed* usan una sola pila de ejecución para mantener el estado de ejecución de ambos tipos de código, al existir casos en los que código no *managed* debe llamar y usar código *managed*, como:

- Gran parte del motor de ejecución está escrito con código no *managed* por motivos de eficiencia, pero el *JIT* tiene una serie de funciones auxiliares (*helpers*) y necesita usar partes de la librería básica de clases (ambos *managed*) que deben ser llamadas desde el código compilado.
- Componentes *managed* pueden ser creados y sus servicios usados por código nativo que trabaje con el *CLI*.
- Algunos hilos pueden unirse a la ejecución desde "fuera".

Dado que cualquier componente puede llamar a rutinas externas mediante el estándar *ECMA P/Invoke* [MSDNPInvoke06], y se puede pasar cualquier tipo de referencia como parámetro en este tipo de llamadas, el entorno de ejecución puede recibir llamadas a los componentes desde fuera del entorno de ejecución. Además cualquier aplicación podría tener una instancia del motor de ejecución y llamar desde la misma a instancias de clases existentes en dicho motor. Para permitir este tipo de llamadas asíncronas a componentes, el modelo de concurrencia de *SSCLI* es bastante complejo. Por otra parte, la forma de crear un hilo y su puesta en marcha es un proceso también complejo que no se va a explicar aquí.

18.2.9.1 MANIPULACIÓN DE LA PILA EN EL MOTOR DE EJECUCIÓN

Una vez una instancia de un hilo ha sido asociada con su hilo de la *PAL*, el código *managed* puede ser ejecutado en él, por ejemplo, mediante el uso del método `Call` de la clase `MethodDesc`. La implementación de este método se apoya en dos estructuras muy importantes existentes en la pila antes de que el código sea procesado por el *JIT*:

- **Un manejador de excepciones**, que envolverá al código *managed*.
- **Una cadena de marcos (*frames*) del motor de ejecución**. Estos marcos serán utilizados para anotar partes de la pila con información en tiempo de ejecución producida por el motor de ejecución.

Los *frames* del motor de ejecución no existen en aquellas partes de la pila generadas por el compilador *JIT*, ya que el motor de ejecución tiene un conocimiento

pormenorizado acerca de cómo el código asociado va a usar la pila, y puede leer y tratar toda la información directamente, sin que se usen este tipo de marcos para anotar ningún dato. No obstante, la existencia de diferentes convenios de llamada, excepciones y otros detalles que deben tenerse en cuenta cuando se interprete el contenido de una pila para el código no producido por el *JIT* (o bien perteneciente a sus *helpers*) hace que el uso de *frames* sea completamente imprescindible para el *SSCLI*. Todo el control de la pila y demás servicios relacionados son mantenidos por el módulo *code manager*.

Teniendo en cuenta lo dicho en el párrafo anterior, debe quedar claro que en la pila de ejecución del *SSCLI* hay mucha más información que el estado de la invocación de un método. En realidad, la pila de ejecución se usa, entre otras cosas, para:

- Mantener y actualizar las referencias a objetos guardados en la pila, para permitir la recolección de basura.
- Mantener información de estado que permita chequeos de seguridad.
- Reconocer transiciones, como las existentes entre dominios o código *managed* y no *managed*.
- Encontrar el manejador correcto y tratar la pila correctamente durante la ejecución de una excepción.
- Generar trazas que puedan ser usadas por el depurador y las excepciones, de manera que estas trazas sean legibles por el programador.

18.2.10 Excepciones

18.2.10.1 MODELO DE EXCEPCIONES

El modelo de excepciones que posee el *CLI* es el mecanismo tradicional basado en bloques `try`, `catch` y `finally`, junto con sentencias `throw`, que se puede encontrar en otros lenguajes como *Java*. La especificación *ECMA* correspondiente describe detalladamente el modelo abstracto para las excepciones; Existen cuatro tipos distintos de bloques: `filter`, `catch`, `fault` y `finally`, varios *opcodes* dedicados a asegurar que el flujo de control pueda ser verificado y un amplio árbol de tipos de excepciones. Usando esos bloques es posible construir las distintas sintaxis de excepciones de alto nivel, usadas en los diferentes lenguajes que soporta el *CLI* y transformadas luego en un modelo común a nivel de *CIL*. Éstos son los *opcodes* usados para controlar excepciones:

Opcode	Uso
`Throw`	Lanza una excepción, usando el objeto en la pila
`Rethrow`	Relanza una excepción dentro de un bloque `catch`
`Leave`	Usado dentro de una sección de código protegido para saltar a un punto específico. El salto causará la ejecución de todos los manejadores de finalización existentes
`Endfinally` (o `endfault`)	Usado al final de un manejador de finalización para devolver el control al motor de ejecución

`endfilter`	Usado al final de una función *filter* para devolver el estado del *handler* de nuevo al motor de ejecución

En el modelo *CLI*, un bloque de código puede estar protegido por uno o más manejadores de excepción. A este tipo de bloques se les denominan bloques protegidos (*guarded blocks*). Dentro de un mismo marco de pila pueden existir múltiples bloques protegidos. Es bastante común tener múltiples manejadores para un mismo bloque, cubriendo cada uno de ellos un subconjunto de posibles excepciones. Los manejadores pueden contener bloques `catch` y/o `finally`, y los bloques `catch` pueden contener filtros asociados con ellos que permite especificar un código que determine si el bloque `catch` es elegido para capturar una determinada excepción o no.

Internamente, los manejadores de excepción se agrupan juntos en una tabla de información de excepciones (*Exception Information Table*, *EIT*). Los ensamblados contienen información comprimida acerca de la *EIT* dentro de las cabeceras que describen métodos. Esta tabla comprimida contiene *offsets* que designan lugares dentro del código *CIL*, y el *JIT* la traduce a la tabla definitiva en tiempo de ejecución, que se localiza en memoria justo encima del código nativo compilado para cada método.

Usando la *EIT* que corresponde al método para cada registro de activación en la pila, el motor de ejecución puede determinar qué manejadores estarán activos en cualquier punto de la ejecución. Cuando una excepción se lanza, todos los manejadores `catch` se "visitan" por orden. Cada uno de ellos debe elegir entre manejar la excepción o ignorarla. Cuando una excepción es manejada por uno de estos bloques, antes de retornar el control, los manejadores `finally` y `fault` pueden ser ejecutados para hacer las tareas de finalización requeridas. Los bloques `finally` siempre son ejecutados, mientras que los `fault` se llaman sólo si la excepción se ha originado en el bloque que guardan. Estas estructuras de datos se emplean para construir la tabla *EIT*:

```
typedef struct IMAGE_COR_ILMETHOD_SECT_FAT
{
    unsigned Kind : 8;
    unsigned DataSize : 24;

} IMAGE_COR_ILMETHOD_SECT_FAT;

// defintitions for the Flags field below (for both big and small)
typedef enum CorExceptionFlag
{
    COR_ILEXCEPTION_CLAUSE_NONE,                    // This is a typed handler
    COR_ILEXCEPTION_CLAUSE_OFFSETLEN = 0x0000,      // Deprecated
    COR_ILEXCEPTION_CLAUSE_DEPRECATED = 0x0000,     // Deprecated
    // If this bit is on, then this EH entry is for a filter
    COR_ILEXCEPTION_CLAUSE_FILTER  = 0x0001,
    COR_ILEXCEPTION_CLAUSE_FINALLY = 0x0002,        // This clause is a finally clause
    // Fault clause (finally that is called on exception only)
    COR_ILEXCEPTION_CLAUSE_FAULT = 0x0004,
} CorExceptionFlag;


typedef struct IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT
{
    CorExceptionFlag    Flags;
    DWORD               TryOffset;
    DWORD               TryLength;      // relative to start of try block
    DWORD               HandlerOffset;
    DWORD               HandlerLength;  // relative to start of handler
    union {
        DWORD           ClassToken;     // use for type-based exception handlers
        // use for filter-based exception handlers (COR_ILEXCEPTION_FILTER is set)
        DWORD           FilterOffset;
    };
} IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT;

typedef struct IMAGE_COR_ILMETHOD_SECT_EH_FAT
```

```
{
    IMAGE_COR_ILMETHOD_SECT_FAT   SectFat;
    IMAGE_COR_ILMETHOD_SECT_EH_CLAUSE_FAT Clauses[1];     // actually variable size
} IMAGE_COR_ILMETHOD_SECT_EH_FAT;
```

Estas tablas pueden existir para cada método. Usando el *array Clauses*, un número variable de manejadores pueden existir también para cada método. Cada entrada en la tabla tendrá un bloque protegido, representado por *TryOffset* y *TryLength*, seguido por un bloque de manejadores, representado por *HandlerOffset* y *HandlerLength*.

El modelo de excepciones propuesto por el *CLI* debería poder interoperar con el modelo de excepciones nativo de *Windows* (*SEH*), para permitir que tanto el código *managed* y *unmanaged* puedan compartir la misma pila en ejecución. No obstante, esto es un problema de cara a la portabilidad del código, ya que algunas plataformas a las que se portará *SSCLI* no comparten el mismo modelo de excepciones de *Windows* o incluso no tienen soporte para un modelo de excepciones a nivel de sistema, lo que complica esta tarea. Para solucionarlo, se incluye un mecanismo de excepciones portable como parte de la *PAL,* usando el *interface* del *SEH* de *Windows* como base, siendo únicamente estas operaciones las que deberán ser portadas. Además, la implementación de alto nivel y la implementación a bajo nivel pueden lanzarse excepciones mutuamente y ser capturadas por la otra parte sin restricciones. Esto implica por ejemplo, que si en las ampliaciones que se van a realizar al *SSCLI*, internamente el código añadido al núcleo del sistema lanza una excepción, esta excepción podrá ser perfectamente capturada por un programa *C#* (o cualquier otro lenguaje de alto nivel) sin problemas.

18.2.10.2 LANZAMIENTO DE EXCEPCIONES

Las excepciones *hardware* o *software* pueden tener su origen desde el código de usuario que ha sido compilado o bien desde dentro del motor de ejecución, pero en todos los casos se manejan según un modelo uniforme de tratamiento para el programador de un lenguaje de alto nivel. Para permitir esto, toda excepción que se lanza (independientemente de quien lo hace) se maneja a través de la capa *SEH* de la *PAL* mencionada anteriormente. Existen tres formas diferentes de lanzar excepciones durante la ejecución de código *managed*, producido por el compilador *JIT*:

- Programáticamente, desde código *managed* que lanza una excepción.
- Programáticamente, desde el código *C++* que implementa el motor de ejecución.
- Por el *hardware* directamente, no iniciado por el *software*.

Si el origen es el código *managed*, se crea una instancia de un objeto adecuado y se ejecuta la función interna *JIT_Throw*, que convierte la excepción a una forma compatible con la capa portable *SEH* descrita, lanzándola usando esta capa de bajo nivel.

```
HCIMPL1(void, JIT_Throw,  Object* obj)
{
    THROWSCOMPLUSEXCEPTION();

    /* Make no assumptions about the current machine state */
    ResetCurrentContext();

    FC_GC_POLL_NOT_NEEDED();    // throws always open up for GC
    HELPER_METHOD_FRAME_BEGIN_ATTRIB_NOPOLL(Frame::FRAME_ATTR_EXCEPTION);    // Set up a frame

    VALIDATEOBJECTREF(obj);
```

```
#if defined(_DEBUG) && defined(_X86_)
    __helperframe.InsureInit();
    g_ExceptionEIP = __helperframe.GetReturnAddress();
#endif // defined(_DEBUG) && defined(_X86_)

    if (obj == 0)
        COMPlusThrow(kNullReferenceException);
    else
        RaiseTheException(ObjectToOBJECTREF(obj), FALSE);

    HELPER_METHOD_FRAME_END();
}
HCIMPLEND
```

El proceso seguido es el siguiente: la llamada a la función *ResetCurrentContext* reinicia el *hardware*, permitiendo reiniciar también la *FPU* y ciertos valores de máscaras. Después de esto, se inicia una transición entre código *JIT* y código del motor de ejecución. Este salto se realiza de forma que tanto el módulo de seguridad como el recolector de basura sean conscientes de dicha transición. Tras esto la excepción se lanza usando *RaiseTheException*.

Cuando la excepción se lanza desde dentro del motor de ejecución directamente en lugar de desde el código *JIT*, existen una serie de funciones que se encargan del lanzamiento de excepciones. La más usada es *COMPlusThrow*, que procesa el lanzamiento de la excepción asegurándose que el recolector de basura está en un estado correcto, y que el entorno de ejecución está capacitado para ejecutar código *managed* (el entorno puede no haber sido inicializado del todo cuando la excepción se produce). En este último caso, no se intentará lanzar una excepción *managed,* ya que es imposible que sea capturada. Si el error es ocasionado por no haber más memoria disponible en el motor de ejecución, se lanzarán excepciones ya alojadas en memoria preparadas específicamente para este caso. Si no, la instancia de la excepción *managed* se crea y se procesa normalmente.

18.2.10.3 MANEJO DE EXCEPCIONES

Dentro del entorno de ejecución el manejo de excepciones se realiza mediante un proceso en dos pasos. En el primer paso, se hace un recorrido de la pila que chequea todos los registros de activación de los métodos existentes en la misma, hasta encontrar un manejador adecuado. En el segundo paso, se buscan todos los bloques `finally` en la región de la pila que se va a liberar para llamarlos antes de que la ejecución `continue`.

Para iniciar este recorrido en dos pasos, el motor de ejecución prepara las regiones correspondientes a código *managed* instalando un manejador estándar de excepciones vinculado con cada una de estas regiones. De esta forma, siempre existirá un manejador *SEH* que controle el procesamiento de la misma, que siempre tendrá también la opción de manejar primero la excepción lanzada sin importar el origen de la misma. El mecanismo concreto y detallado del procesamiento de excepciones en *SSCLI* es más extenso del mostrado, y puede seguirse con más detalle en [Stutz03].

18.2.11 Manejo de Memoria en el Entorno de Ejecución: Recolección de Basura

SSCLI posee un recolector de basura para la manipulación de la memoria

empleada por los programas. A la hora de hacer cualquier modificación al sistema debemos entonces tener muy en cuenta su existencia y funcionamiento, construyendo el nuevo código para no entrar en conflicto con el mismo. Por ello describiremos brevemente el funcionamiento de este recolector.

SSCLI posee los tres mecanismos de reserva de espacio en memoria típicos de un programa *C* o *C++*: **Estática**, **dinámica** y **de pila**. Tanto en el primer caso como en el tercero, no es necesaria la incorporación de un mecanismo de recolección de memoria, ya que no tiene sentido. No obstante, estos dos mecanismos no cubren todas las necesidades del programador, y es por ello que es preciso incorporar un modelo de memoria dinámica, que le permita reservar a voluntad zonas de memoria para manipularlas como desee, dentro de una zona de memoria llamada *heap*. Al ser creadas por un programador cuando lo considere oportuno, éste tiene también la responsabilidad de eliminarlas cuando dejen de ser necesarias, salvo que se incorpore un mecanismo de recolección de basura como el que posee *SSCLI*. El uso incorrecto de este tipo de memoria dinámica es una de las principales fuentes de error de los programas, por lo que la incorporación de un recolector de basura elimina una importante fuente de errores, al no tener que controlar el programador este tipo de operaciones. De esta forma, el programador simplemente pide memoria, mientras que el entorno de ejecución controla su uso y el recolector de basura determina cuando se puede liberar esa memoria, haciéndolo en el momento oportuno.

El uso de un recolector de basura tiene otros beneficios añadidos en caso de que el código ejecutado tenga errores o sea malicioso. En cualquier caso, poseer un recolector de basura impide que el programador pueda acceder directamente a direcciones de memoria, salvo en determinados momentos y de forma muy controlada. En cuanto al empleo del recolector de basura en los programas, aunque éste se ocupa de la gestión de la memoria, los programas usan otros recursos (como archivos, ventanas, *sockets*, etc.) que pueden estar fuera del motor de ejecución y el recolector no puede manejarlos, ya que normalmente su "dueño" es el sistema operativo. Si un tipo está pensado para representar a un recurso externo (por ejemplo un archivo), entonces este tipo debe ser el encargado de adquirir y liberar el recurso al que representa cuando sea necesario. No obstante, liberar un recurso de esta clase cuando se depende de un recolector de basura no es una tarea sencilla, ya que el programador ya no tiene la responsabilidad de liberarlo y el recolector lo eliminaría de memoria sin, en teoría, hacer ninguna operación adicional. No obstante, el *SSCLI* sí que posee mecanismos para estos casos, siendo uno de ellos la finalización. Mediante este mecanismo, el recolector llama al método `Finalize` cuando el objeto va a ser liberado, siempre que el objeto en sí implemente alguna funcionalidad para el mismo. Este método permite que un objeto, antes de ser finalmente liberado, ejecute algunas operaciones que, entre otras cosas, puedan liberar recursos externos, ya que el *SSCLI* asegura que será llamado antes de la eliminación del objeto de memoria. No obstante, este mecanismo no permite al programador liberar recursos en un momento concreto de la ejecución.

18.2.11.1 El Heap en el SSCLI

Una simplificación de un proceso de recolección de basura consistiría en la reserva de una determinada cantidad de memoria para formar el *heap* del programa. Dentro de este *heap*, el programador reserva bloques de memoria adecuados para los objetos que necesita y el recolector de basura eliminará aquéllos que ya no estén en uso, algo que puede hacerse cuando los objetos ya no tengan más referencias apuntándolos, ya que el recolector lleva una contabilidad adecuada de las mismas para cada objeto.

El proceso seguido para esta tarea en el *SSCLI* consiste en la renovación periódica del *heap*, identificando los objetos que no están en uso y liberándolos posteriormente, construyendo una lista de zonas de memoria adecuadas para su uso posterior. La

primera parte de este proceso es la llamada *tracing*. Mediante el mismo, el recolector sigue y registra todas las referencias "vivas" (que apuntan a un objeto válido y en uso) en el *heap*, a partir del contenido de todas las pilas de memoria en ejecución, la memoria estática y otros lugares concretos. Una vez se sigue un puntero, se examina el contenido de la memoria a la que apunta en busca de más punteros, siendo éstos a su vez examinados con el mismo criterio, y así sucesivamente. De esta forma, el recolector puede localizar fácilmente cualquier zona de memoria que todavía esté en uso y no eliminará ninguna zona inadecuada.

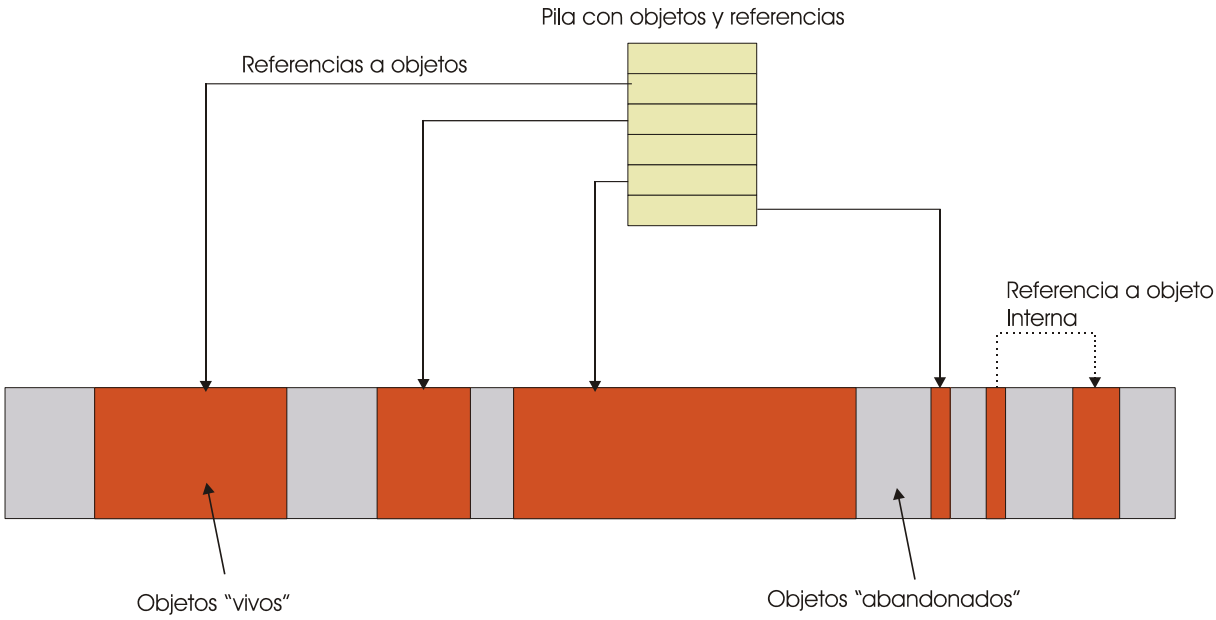

Figura 18.6: Seguimiento de zonas de memoria libres

Una vez realizada esta tarea, como ya se ha mencionado una de las tareas más sencillas a realizar es simplemente crear una lista de zonas de memoria libre, lista a partir de la cual se obtendrán bloques de memoria para usar posteriormente. A este proceso simple se le denomina "*mark and sweep*". No obstante, esto puede causar una elevada fragmentación de la memoria a medida que se usa, por lo que se debe incorporar un proceso de compactado de memoria que elimine el problema. Este proceso de compactado consiste en mover toda la memoria que continúa siendo usada a las direcciones más bajas de cada segmento del *heap*, modificando luego la dirección a la que apuntan los punteros correspondientes de forma acorde, manteniendo la coherencia del sistema. En la siguiente figura se ve como queda la memoria tras este proceso:

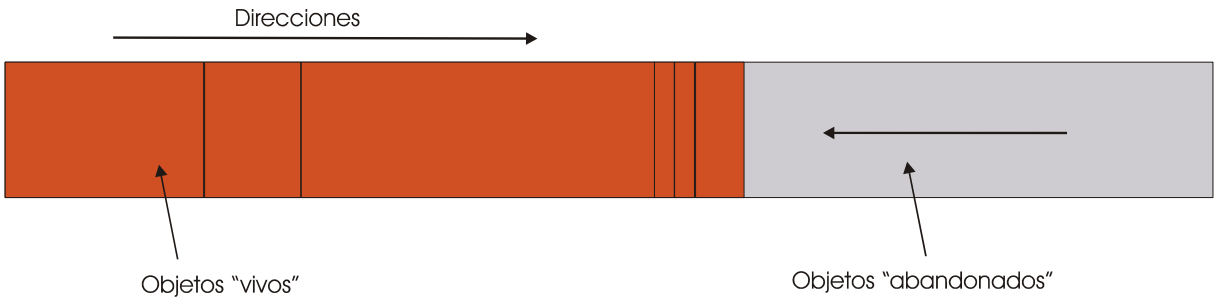

Figura 18.7: Proceso de compactación de memoria

Poner a todas las zonas de memoria en uso juntas tiene además otro tipo de

ventajas, al aumentar la localidad de las referencias, entre otros beneficios. *SSCLI* no usa este mecanismo de compactación simple visto, sino uno llamado "*copying collection*", consistente en copiar todos los objetos "vivos" a un *heap* nuevo, tras la cual el *heap* "antiguo" es eliminado o reutilizado. Las ventajas de esta técnica son principalmente una mayor simplicidad a la hora de buscar espacio para los nuevos objetos y también en la gestión de la memoria en general. No obstante, a la hora de modificar las referencias para que apunten a las nuevas localizaciones de los objetos a los que apuntaban puede haber algún coste adicional, además de tener que ocupar el doble de memoria.

Un refinamiento de esta técnica, que reduce los costes de la copia de los objetos es la llamada "*generational collection*", en la que se dividen los objetos en "generaciones", marcados por el tiempo que lleven activos. Esta técnica explota el hecho de que los objetos pueden permanecer "vivos" un tiempo diferente en memoria, ya que alguno pueden ser de vida muy corta o muy larga en función de cómo se usan y su función concreta y además también pueden tener diferentes tamaños. Lo que se hace entonces es dividir el *heap* en zonas que están pensadas para contener objetos de características similares en cuanto a su tiempo de vida, y procesar estas zonas usando algoritmos que exploten mejor estas características, usando la técnica más adecuada para procesar dichos objetos y reduciendo el coste por dos vías:

- No procesando toda la memoria al mismo tiempo, ya que las diferentes zonas son examinadas en diferentes momentos y con diferentes frecuencias.
- El algoritmo utilizado es el más adecuado al tiempo de vida de los objetos, logrando la máxima eficiencia a la hora de su recolección.

La técnica utilizada hace que al principio todos los objetos estén ubicados en la generación más joven y, si "sobreviven" a un ciclo de recolección, se pasan a la siguiente generación mediante una copia. Al final, los objetos en la generación más joven tienen un tiempo de vida menor y los objetos en la generación siguiente tienen uno mayor. Con esta división, pueden usar diferentes técnicas de recolección para los objetos en distintas generaciones. Por ejemplo, una estrategia basada en evitar las copias y la compactación funciona mejor para la generación mayor, mientras que para la generación más joven es mejor usar una estrategia de compactación y copia, dada su diferente naturaleza.

Esta última técnica es la utilizada por el *SSCLI*, empleando dos generaciones como las vistas. El recolector se activa por la falta de memoria en un momento concreto, de forma que cuando esta carencia de recursos se detecte, una o las dos generaciones son procesadas como se ha descrito anteriormente. El proceso de gestión de memoria en el *SSCLI* contempla más aspectos que los aquí descritos, y para más detalles se recomienda acudir a la bibliografía de referencia [Stutz03]. En lo referente a esta tesis, debemos tener en cuenta que la existencia de un recolector de basura hace que debamos tener especial cuidado con los objetos que creemos como parte de las modificaciones planteadas al sistema, ya que debemos integrar éstos con el recolector de basura para que sean reciclados correctamente y no interfieran con el mismo. También debemos tener especial cuidado para que los objetos que necesitemos crear no sean reclamados por el recolector mientras los estemos usando, por lo que debemos usar los mecanismos adecuados para que el recolector los identifique correctamente como "en uso".

18.2.12 Proceso de Arranque del Sistema: Ejecución de un Programa

A continuación se describirá el proceso de ejecución de un código que esté dentro de un ensamblado. Este código no puede ser ejecutado hasta que el ensamblado se localiza, se carga, se verifica y compila. El *CLI* es también un código contenido en diferentes ensamblados, que deben ser cargados y ejecutados. Por ello, siempre tiene que haber un ensamblado primario por defecto, que tiene un punto de entrada especial donde el proceso de carga y comprobación de todo el resto de código comienza. El proceso de arranque de un ejecutable en *SSCLI* se hace a través del programa *clix*, que obtendrá el ejecutable a arrancar como parámetro. El hecho de tener que ejecutar un programa concreto para arrancar código ejecutable permite crear un *clix* para cada una de las plataformas y de esta forma poder ejecutar el mismo código en ellas, permitiendo también tener múltiples versiones de *clix* funcionando al mismo tiempo.

El código encargado de cargar el ensamblado también lo integra dentro del *CLI*, lo ejecuta y el código de retorno resultante de su ejecución se devuelve al sistema operativo. El proceso de carga se puede dividir en los siguientes pasos, que se pueden ver en la función `Launch` del fichero *clix.cpp*:

- Se registra la librería *rotor_palrt* usando la función `PAL_RegisterLibrary`. Las librerías *rotor_pal* y *rotor_palrt* se combinan para crear la implementación de la *PAL* que es necesaria para ejecutar el *SSCLI*.
- Se obtiene el nombre del ensamblado que se debe cargar en el *CLI*. En *clix* se obtiene de la línea de comandos.
- Obtienen el nombre del motor de ejecución que va a ser cargado. En el caso de *clix*, esto se obtiene a partir del *path* completo del fichero *clix.exe*.
- Carga la librería *sscoree* y obtiene un puntero a la función `_CorExeMain2`.
- Se ejecuta `_CorExeMain2` junto con el archivo del ensamblado que va a ser cargado, dando el control completo del proceso al *CLI*.

Una vez llegado a este punto, el proceso de carga está casi completo, aunque posteriormente se hacen algunas operaciones adicionales, como chequeos de seguridad.

19 Apéndice B: Especificación de la Interfaz de Operaciones de las Clases Desarrolladas

Tras la labor de diseño realizada en capítulos anteriores, se mostrará ahora la interfaz pública de las nuevas clases creadas para implementar las funcionalidades descritas. Para la implementación de estas funcionalidades han tenido que llevarse a cabo multitud de cambios a muchas partes del sistema, para eliminar comprobaciones en momentos concretos que impedían llevar a cabo diferentes tareas, integrar las nuevas clases en el sistema existente o bien permitir el acceso a determinadas funcionalidades para ser usadas por nuestros servicios, pero no serán mostrados aquí dado su gran número. El código fuente del sistema extendido está completamente comentado en estos puntos para poder así consultarlos en caso de necesidad y saber qué se ha hecho en cada momento. Por ello, mostraremos sólo ahora las clases creadas nuevas, divididas en paquetes y por su nivel, separando las clases de alto nivel (*BCL*) de las implementadas dentro del motor de ejecución.

19.1 Clases de la librería BCL

19.1.1 Paquete `System.Reflection.Structural`

Este paquete se ha creado nuevo para contener la mayoría de las clases que tienen la funcionalidad reflectiva de los diferentes pasos o *steps* de la implementación.

19.1.1.1 Clase *FieldCollection*:

Esta clase se usa para contener información de atributos en el *Paso 1* de la implementación del sistema, creando una lista de ellos que se puedan asociar a cualquier objeto o clase.

```
namespace System.Reflection.Structural
{
        using System.Collections;
        using System.Reflection;

        [Serializable()]
        public class FieldCollection : IMemberCollection
        {
```

```
                private Hashtable fields;
                const int INITIAL_SIZE = 10;
                const float INITIAL_LOAD_FACTOR = 0.7f;


                public FieldCollection ();
                public void addMember (Object member);
                public void removeMember (Object member);
                public Object getMember (Object member);
                public void setMember (Object member);
                public void setMemberValue (string name, Object val);
                public bool existMember (Object member);
                public bool existMemberByName (string name);
                public ICollection enumerate ();
                public int Count ();
        }
}
```

19.1.1.2 CLASE *IMEMBERCOLLECTION*:

Esta interfaz es común a todas las clases que se usan para guardar listas de miembros, para procurar tener un tratamiento homogéneo de las mismas.

```
namespace System.Reflection.Structural
{
        using System.Collections;

        public interface IMemberCollection
        {
                void addMember (Object member);
                void removeMember (Object member);
                Object getMember (Object member);
                void setMember (Object member);
                bool existMember (Object member);
                ICollection enumerate ();
        }
}
```

19.1.1.3 CLASE *METHODCOLLECTION*:

Esta clase se usa para guardar listas de métodos asociadas a clases u objetos, al igual que la lista anterior para atributos.

```
namespace System.Reflection.Structural
{
        using System.Collections;
        using System.Reflection;

        [Serializable()]
        public class MethodCollection : IMemberCollection
        {
                private Hashtable methods;
                const int INITIAL_SIZE = 10;
                const float INITIAL_LOAD_FACTOR = 0.7f;


                public MethodCollection ();
                public void addMember (Object member);
                public void removeMember (Object member);
                public Object getMember (Object member);
                public void setMember (Object member);
                public void setMemberValue (string name, Object val);
                public bool existMember (Object member);
                public bool existMemberByName (string name);
```

```
                public ICollection enumerate ();
                public int Count ();
        }
}
```

19.1.1.4 CLASE *METHODWRAP*:

Esta clase se usa para guardar la información de un método, y se usa para crear métodos en tiempo de ejecución.

```
namespace System.Reflection
{
        using System;
        using System.Reflection.Structural;

        //This class wraps an added method to an object.
        public class MethodWrap
        {
                public Object Owner;
                //This attribute stores the result of _mInfo.ToString().
                public string Signature;
                /*The original type was MethodInfo, but was changed in order to solve mscorlib
                type definition constants restrictions. If this attribute has a non null value,
                then is used to redirect the Invoke calls. If not, its supposed that this class
                has a method defined who matches the invoked one (using inheritance). However,
                this second technique only works in the BCL - based implementation, which is
                incompatible with the native one (different rotor installments must be provided
                to work separately with both distribution.*/
                private Object _mInfo;
                //This flag indicates if the wrapped method has been "deleted" or not.
                private int _IsDeleted;
                public String Sig;
                public MethodInfo mInfo;
                public bool IsDeleted;

                public MethodWrap ();
                public MethodWrap (bool deleted, string sig);
                public MethodWrap (MethodInfo mI);
                public Object Invoke (string name, Object [] param);
                public override bool Equals (Object obj);
                public string getSignature ();
                public override String ToString ();
        }
}
```

19.1.1.5 CLASE *STRUCTURAL*:

Esta clase centraliza todos los servicios reflectivos e información reflectiva del *Paso 1* de nuestra implementación, permitiendo que se pueda acceder a estos servicios desde cualquier lenguaje, al pertenecer a la *BCL*.

```
namespace System.Reflection.Structural
{
        using System;
        using System.Reflection;
        using System.Collections;
        using System.Reflection.Emit;
        using System.Reflection.Disassembly;

        sealed class AssemblyProvider : IAssemblyProvider
        {
                public Assembly Load(string assemblyName);

                public Assembly[] GetAssemblies();
```

```
}

public class Structural
{
        //Tabla de objeto - Campos añadidos o sobrecargados en el objeto.
        private static Hashtable fieldMap = new Hashtable(20, 0.7f);
        //Tabla de clase - Campos añadidos o sobrecargados a la clase.
        private static Hashtable classFieldMap = new Hashtable(20, 0.7f);
        //Metodos de objetos.
        private static Hashtable methodMap = new Hashtable(20, 0.7f);
        //Metodos de clase.
        private static Hashtable classMethodMap = new Hashtable(20, 0.7f);
        private static AssemblyName an;
        private static AppDomain ad;
        private static AssemblyBuilder ab;
        private static ModuleBuilder mbld;
        private static int typeCount = 0;

        //Obtiene (si existe) el campo cuyo nombre se indica para el objeto o clase
        //pasado.
        public static FieldInfo getField (Object owner, string name);
        //Este método debe ocuparse de encontrar si un determinado objeto o clase tiene
        acceso a un determinado atributo en un momento dado.
        private static Object internalGetField (Object owner, string name);
        //Obtiene el valor de un atributo en un momento dado.
        public static Object getFieldValue (Object owner, string name);
        public static void addField (Object owner, RuntimeStructuralFieldInfo rsfi);
        //Busca un atributo añadido en tiempo de ejecucion en la tabla especificada
        para el objeto dado.
        private static RuntimeStructuralFieldInfo searchStructuralField(Object owner,
                                                Hashtable table, string name);

        //Busca un atributo que haya sido declarado en tiempo de compilación en la
        //clase.
        private static FieldInfo searchDeclaredField(Object owner, string field);
        private static RuntimeStructuralFieldInfo Field2Structural (Object owner,
                                                                FieldInfo f);
        //Un objeto no puede borrar campos de su clase, pero una clase puede de los
        suyos y de los heredados.
        public static void removeField (Object owner, string name);
        public static void alterField (Object owner, RuntimeStructuralFieldInfo rsfi);
        public static void setField (Object owner, string name, Object val);
        public static bool existField (Object owner, string name);
        private static bool haveFieldDirectly (Object owner, string name);
        public static void clear (Object owner);

        /*************************************************************************/

        private static void initMethodSupport ();
        private static void addExElement(Hashtable h, string label, Object type);
        /**
         * Este metodo debe:
         * - Tomar todos los datos del metodo proporcionado.
         * - Hacer la copia y reasignacion del this al ultimo parametro del mismo
               (calculando a cual corresponde).
         * - Crear un objeto cuyo contenido sea dicho metodo.
         * - Devolver dicho objeto.*/
        private static MethodWrap copyMethodFor(Object target, MethodInfo method);
        private static OpCode determineThisPos (MethodInfo m, out int pos);
        public static MethodWrap getMethod (Object owner, Type ret, string name, Object
                                            [] param);
        public static string parsePackage(string sig);
        public static string createSignature (string retType, string name, Object []
                                            param);
        public static string signature2Name (string sig);
        /*Este metodo obtendra un metodo declarado dentro de las estructuras, primero
        mirando los objetos y luego las clases, a partir de su signatura.*/
        private static Object internalGetMethod (Object owner, string sig);

        /**
         * Añade el metodo al objeto dado. Para ello debe:
         * - Tomar el metodo y copiarlo con el metodo correspondiente.
         * - Crear un par (owner, lista de metodos), añadiendo el metodo a dicha lista.
         * - En la lista se añadira automaticamente un par (nombre completo (ToString),
         * Objeto generado con el metodo).
         * NOTA: Los objetos generados automaticamente tendran las siguientes
         * propiedades:
```

```
                * - IsDeleted: Indicara si el metodo ha sido borrado. Usable cuando se
                * enmascare uno existente.
                * NOTA 2: El criterio a seguir para la precedencia de los metodos sera:
                * a) Clases y objetos mantendran listas separadas de metodos.
                * b) La busqueda se hara primero por los objetos, luego su clase y por ultimo
                * la clase padre.
                * c) Un objeto puede añadir un metodo de forma privada al mismo, aunque se
                * llame igual que el de la clase (lo enmascara).
                * d) Se evita que los elementos se repitan dentro de la lista de cada clase u
                * objeto. */
               public static void addMethod (Object owner, MethodInfo miP);
               //Busca un atributo añadido en tiempo de ejecucion en la tabla especificada
               //para el objeto dado.
               private static MethodWrap searchStructuralMethod(Object owner, string sig);
               private static MethodWrap searchStructuralMethod(Object owner, Hashtable table,
                                                   string ret, string name, Object [] param);
               //Busca un metodo que haya sido declarado en tiempo de compilación en la clase.
               private static MethodWrap searchDeclaredMethod(Object owner, string sig);
               private static MethodWrap searchDeclaredMethod(Object owner, string ret, string
                                                       name, Object [] param);
               public static void removeMethod (Object owner, MethodInfo mI);
               //Un objeto no puede borrar campos de su clase, pero una clase puede de los
               suyos y de los heredados.
               public static void removeMethod (Object owner, string ret, string name, Object
                                                   [] param);
               public static void alterMethod (Object owner, MethodInfo or, MethodInfo val);
               public static bool existMethod (Object owner, string ret, string name, Object
                                                   [] param);
               private static bool haveMethodDirectly (Object owner, string ret, string name,
                                                   Object [] param);
               public static Object invoke(Object target, string ret, string mName, Object []
                                                   param);
       }
}
```

19.1.2 Paquete `System.Reflection`

Se han añadido las siguientes clases a este paquete ya existente:

19.1.2.1 Clase *RuntimeStructuralFieldInfo*:

Esta clase se usa para contener la información de los atributos nuevos creados, que serán añadidos a clases u objetos posteriormente empleando nuestras primitivas de reflexión estructural.

```
namespace System.Reflection {
       /**
        * This class is provided as a appropiate method to store information about a
        * dinamically added field. When a field is added to an object or class, an instance of
        * this class must be provided with the necessary information about the field in order
        * to be recovered accordingly later. Note that this class is a child of the
        * RuntimeFieldInfo class, which is used to store information about a coded field. This
        * ensures us a reasonable capability to treat both types of field in an uniform way
        * during execution, conecting our implementation to the current infraestructure of the
        * Rotor without too many disruptions.*/
       [Serializable()]
       public class RuntimeStructuralFieldInfo : RuntimeFieldInfo, ISerializable
       {
              /**
               * Name of the field.
               */
              private string _name;
              /**
```

```
 * Owner of the field (object or class).
 */
private Object _owner;
/**
 * Type of the field.
 */
private Type _type;
/**
 * Current value of the field (always an object subclass). Autoboxing happens
 * when a value type object is provided.*/
private Object _value;
/**
 * Attribute which store a number indicating which is the type of this field
 * (see SignatureHelper for details about the possible values). This field is
 * automatically handled by the object when the user stores a value.*/
private int valueSelect = 0;
/**
 * Attributes of the field. This field is coded as an int to avoid memory
 * alignment problems with the unmanaged side of this class. */
private int _attributes;
/**
 * Different flags used in our model.
 */
private bool _isDeleted = false;
private bool _isClass = false;
private bool _isOverload = false;
/* This field allow us to save 2 bytes per instance of this class, because
 * it solves the size mismatch between this type and his native representation,
 * mismatch that occur when one of the classes does not complete a 32 bit
 * aligned size in memory.*/
private bool _foo = false;

/**Default constructor provide default attribute inicialization.*/
public RuntimeStructuralFieldInfo();
/**
 * This constructor is the appropiate way to build a dynamic field, alowing us
 * to initialice all the data which is needed to completely define a field.
 * NOTE: In the current version of the system, the following limitations are
 * present:
 * - The visibility attribute (Public, Protected, Private) is not used for
 * added attributes.
 */
public RuntimeStructuralFieldInfo(string name, Type type, Object val);
/**The same constructor as the above one, but specifying an owner is not
needed.*/
public RuntimeStructuralFieldInfo(Object owner, string name, Type type, Object
                                  val);
public override String Name;
public bool IsDeleted;
public bool IsClass;
public bool IsOverloading;
// Return the class that declared this Field.
public Object Owner;
// Return the string representation of FieldInfo.
public override String ToString();
// Return the class that declared this Field.
public override Type DeclaringType;
// Return the class that was used to obtain this field.
public override Type ReflectedType;
// Return the Type that represents the type of the field
public override Type FieldType;
public override Object GetValue(Object obj);
//These two methods are not applicable in our implementation.
[CLSCompliant(false)]
public override Object GetValueDirect(TypedReference obj);
[CLSCompliant(false)]
public override void SetValueDirect(TypedReference obj,Object value);
public override void SetValue(Object obj,Object val,BindingFlags
                              invokeAttr,Binder binder,CultureInfo culture);
public override RuntimeFieldHandle FieldHandle;
// Return the Attribute associated with this field.
public override FieldAttributes Attributes;
public void SetAttributes(FieldAttributes fieldAttr);

//////////////////////////// ICustomAttributeProvider Interface
// Return an array of all of the custom attributes
public override Object[] GetCustomAttributes(bool inherit);
```

```
                // Return a custom attribute identified by Type
                public override Object[] GetCustomAttributes(Type attributeType, bool inherit);
                // Return if there is a  custom attribute identified by Type
                public override bool IsDefined (Type attributeType, bool inherit);
                public bool isCompatibleWith(RuntimeStructuralFieldInfo obj);
                /**Compares a runtime field with another one, returning false if an object of
                any other class is provided in the parameter.*/
                public override bool Equals(Object obj);
                /**Make a clone of this field.*/
                public RuntimeStructuralFieldInfo Clone ();
    }
}
```

19.1.2.2 CLASE *NATIVESTRUCTURAL*:

Esta clase se ocupa de centralizar todas las operaciones reflectivas de nuestra implementación, sirviendo como *interface* para llamar a la implementación nativa de dichas operaciones en el *Paso 2* de nuestra implementación. Así, la mayoria de estas operaciones estarán vinculadas con un método del nucleo de la máquina implementado en *C*, que se ejecutará de una forma mucho más eficiente.

```
namespace System.Reflection
{
        using System;
        using System.Reflection;
        using System.Runtime.CompilerServices;

        /**
         * This class is the main way to provide us structural reflection for objects and
         * classes. Due to the type restrictions implemented into the C# language, using these
         * methods is the only secure way to work with reflection data. However, the jitter was
         * modified to natively support reflection mechanism, so its possible
         * to incorporate some of the reflection behaviour in a normal C# program without using
         * this library to access/set fields, but taking into account possible type
         * restrictions and other problems that changing a field could arise.
         *
         * This class is a close equivalent to the Structural one, but almost all of his
         * implementation is integrated into the VM, coded in C++, so the speed of the
         * operations greatly improves.
         * Also this class is the perfect way to integrate the support for structural
         * reflection into the Rotor Jitter. The Jitter was heavily modified in order to
         * perform calls to several methods implemented here and to support structral
         * reflection into the CIL language.*/
        [Serializable()]
        public class NativeStructural
        {
                [MethodImplAttribute(MethodImplOptions.InternalCall)]
                public static extern Object getField (Object owner, String name);
                [MethodImplAttribute(MethodImplOptions.InternalCall)]
                private static extern Object getFieldValue (Object owner, String name);


                /* Public method to extract the value of a field. Needed to cope with
                 * reflection-dealing issues when dealing with native fields of a primitive
                 * type.*/
                public static Object getValue (Object owner, String name);

                /**
                 * Redondo: Adds a field to any object or class that is passed as the first
                 * parameter, storing it as part of that element until is deleted. This method
                 * does not allow the user to add a field whose name already exist, no matter
                 * that field is coded or dinamically added, in order to prevent umpleasant
                 * errors or misbehaviours. If an object is provided as the first parameter,
                 * then the field is considered local to that object, so only
                 * that object is able to use that field during execution privately. However,
                 * if a field is added to a class, all objects who are instances of that class
                 * will be able to use a copy of that field during execution, using a lazy
                 * adquisition mechanism that ensure that the object must not have memory
                 * reserved for that field until someone performs an operation over it, saving
```

```
         * memory and CPU cycles when a class is provided. */
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern void addField (Object owner, RuntimeStructuralFieldInfo
                                            rsfi);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern void maskField (Object owner, RuntimeStructuralFieldInfo
                                            rsfi);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern bool auxRemoveField (Object owner, String name, bool
                                            overrideDeleted);
        /**
         * This methods creates a RuntimeStructuralFieldInfo class that match the
         * "coded" field passed. */
        private static RuntimeStructuralFieldInfo Field2Structural (Object owner,
                                                                    FieldInfo f);
        /**
         * The public method used to remove a field allows us to easily mask a field if
         * it can't be physically removed (for example, when the field is coded we
         * cannot alter the object structure in memory).*/
        public static void removeField (Object owner, String name);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern void alterField (Object owner, RuntimeStructuralFieldInfo
                                            rsfi);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern void setFieldValue (RuntimeStructuralFieldInfo field,
                                            Object val);
        /**
         * We distinguish between coded fields and added ones to easily implement this
         * operation.
         */
        public static void setValue (Object owner, String name, Object val);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern bool existField (Object owner, String name);

        /************************ METHOD STRUCTURAL REFLECTION ********************/

        /**
         * This is an interface to the getMethod (object, string) method, which truly
         * executes the operation.
         */
        public static Object getMethod (Object owner, Type ret, String name, Object []
                                    param);
        /**
         * This method ensures that always a MethodWrap object is returned in the call.
         */
        public static Object getMethod (Object owner, String signature);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern Object internalGetMethod (Object owner, String
                                                signature);
        /**
         * This is an interface to the getMethod (object, string) method, which truly
         * executes the operation. */
        public static Object getMethod (Object owner, MethodInfo m);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern void addMethod (Object owner, MethodWrap miP);
        public static void addMethod (Object owner, MethodInfo minf);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern bool auxRemoveMethod (Object owner, String sig, bool
                                            overrideDel);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern void maskMethod(Object owner, MethodWrap mw);
        public static void removeMethod (Object owner, String sig);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        private static extern void alterMethod (Object owner, String sig, Object val);
        /**
         * Parameters:
         * owner - Object which has the method we need to alter.
         * val - MethodInfo of the new method. It's supossed that a method with the
         * same signature is present into the owner and we will alter his code with the
         * new method. For other kind of substitution, use the remove/add combination
         * instead.*/
        public static void alterMethod (Object owner, MethodInfo val);
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern bool existMethod (Object owner, String sig);
        /**
         * This method will enable us to call any method over any object. Errors should
         * not be expected if the method doesn't need a field that the object doesn't
```

```
                 * not have or something similar.*/
                public static Object invoke(Object target, MethodInfo m, ParameterInfo []
                                                paramT, Object [] paramV);
                public static string parsePackage(string sig);
                public static string createSignature (string retType, string name, Object []
                                                param);
                public static string signature2Name (string sig);
        }
}
```

19.2 CLASES DEL MOTOR

A nivel de motor no existe el concepto de paquete, estando organizadas las clases por directorios. Se mostrará sólo el código de las únicas clases añadidas, pertenecientes al directorio *vm*. Adicionalmente, existen multitud de cambios puntuales al motor.

19.2.1 Fichero MethodWrap.h

Esta clase se usa para guardar la información de una método cualquiera en tiempo de ejecución dentro del motor del sistema.

```
class MethodWrap: public Object
{
        public:
                Object *Owner;
                //Al generar los objetos, aquí se debera guardar el toString del MethodInfo del
                // metodo generado.
                StringObject *Signature;
                //Sólo usado en caso de enmnascaramiento de metodos nativos.
                Object *_mInfo;
                INT32 _IsDeleted;

                MethodWrap ();
                ~MethodWrap ();
};
```

19.2.2 Fichero ORHashtable.h

Esta clase es una tabla *hash* que se usa para guardar información de miembros a bajo nivel y poder acceder a ella con la mayor eficiencia posible.

```
class ORHashTable
{
 public:
        ORHashTable();

        int findPos( char *x);
        void makeEmpty();
        void insert(char *x, OBJECTHANDLE o);
        void remove(char *x);
        void remove(int);
```

```
        OBJECTHANDLE find(char *x);
        //REDONDO: Warning->Call only if the specified position have a value!!
        OBJECTHANDLE& operator[](int pos);
        int getExistingElements() {return existing;}

private:

        struct HashEntry
        {
                OBJECTHANDLE elementInfo;
                char *element;

                HashEntry( char *e, OBJECTHANDLE eInfo): element( e ), elementInfo(eInfo);
        };

        int occupied;
        int existing;
        bool isActive(int currentPos) const;
        void rehash( );
        bool isPrime( int n ) const;
        int nextPrime( int n ) const;
        void tolowerstr(char *str) const;

 protected:
        unsigned int hash( char *key) const;
        HashEntry **array;
        int arraySize;
};
```

19.2.3 Fichero RuntimeStructuralFieldInfo.h

Esta clase se encarga de contener la información de un atributo cualquiera a bajo nivel en la implementación de nuestras primitivas dentro del motor del sistema.

```
/* This class is the unmanaged part of the C# class with the same name. It contains a field
   representation of the class in order to use its values into the execution environment with
   ease, using them in order to communicate with the managed part.
   It also contains some operations which were left to the unmanaged part in order to access
   them easily from other modules.*/
class RuntimeStructuralFieldInfo: public ReflectBaseObject
{
        public:
                //See the managed counterpart for descriptions.
                StringObject *_name;
                Object *_owner;
                TypeHandle *_type;
                Object *_value;
                INT32 valueSelect;
                INT32 _attributes;
                INT8 _isDeleted;
                INT8 _isClass;
                INT8 _isOverload;

                /* REDONDO: This method clone a runtime field, which is used when we have to
                "export" a class added field to an object, using the lazy mechanism previously
                described.*/
                RuntimeStructuralFieldInfo *Clone ();
                /* This method is used to test if the field is compatible with the specified
                one. Compatibility is achieved if both fields have the same name and comatible
                (casteable) values. This methods is used when we try to add a field and a
                deleted one is found.*/
                int IsCompatibleWith (RuntimeStructuralFieldInfo *f);
                /* This method create a RuntimeStructuralField info who wraps a coded field
                (represented by a FieldDesc class). Is used when we need to alter or delete a
                coded field, because we cannot physically delete them due to the structure of
                the Rotor environment.*/
                static RuntimeStructuralFieldInfo *WrapFieldDesc (FieldDesc *f);
};
```

19.2.4 Fichero NativeStructural.h

```
/* This class implement all the reflection primitives into the SSCLI VM, interactuating with
   all the necessary Rotor parts to performs all the needed operations. This class is used by
   the NativeStructural managed implementation, which is the only way for the programs to use
   this features, and by the jitter, which uses these primitives to provide CIL reflective
   capacities. See the cpp for method description.*/
class NativeStructural
{
    private:

        //Auxiliar methods.
        static bool isType (Object *owner);
        static OBJECTREF existAddedField (Object *owner, StringObject *name);
        static BOOL internalAuxRemoveField (Object *owner, StringObject *name, BOOL
                                            overrideDeleted);
        static void internalAddField(Object * owner, RuntimeStructuralFieldInfo *rsfi, BOOL
                                     existTest);
    public:

        /************************** FIELD REFLECTION ***************************/

        /*Auxiliar functions. These functions are needed because we cannot invoke directly the
        FCDELCX functions from the rest of VM code, only a caller placed into the managed part
        is allowed to do so.*/
        static FieldDesc * getTrueFieldDesc(Object *owner, const char *name);

        static MethodDesc *NativeStructural::auxGetNativeMethod (Object *owner, LPCUTF8
                                     m_pszDebugMethodName, LPUTF8 m_pszDebugMethodSignature);

        static OBJECTREF NativeStructural::auxExistMethodFromNativeSig (Object *owner, LPCUTF8
                                     m_pszDebugMethodName, LPUTF8 m_pszDebugMethodSignature,
                                     int *isClass, int *isAdded, int *addedDelControl);

        static OBJECTREF NativeStructural::existAddedMethodFromNativeSig (Object *owner,
                                     LPCUTF8 m_pszDebugMethodName, LPUTF8
                                     m_pszDebugMethodSignature);

        static Object *auxGetField (Object *owner, StringObject *name, int *isClass, int
                                     *isAdded, int *addedDelControl);

        static Object *NativeStructural::auxGetFieldValue(Object *owner, StringObject *name,
                                     int checkClass = 0);

        static void NativeStructural::auxSetFieldValue(RuntimeStructuralFieldInfo *field,
                                     Object *val, int checkClass = 0);

        static OBJECTREF auxExistField (Object *owner, StringObject *name, int *isClass, int
                                     *isAdded, int *addedDelControl);

        static int NativeStructural::testForAddedField (Object *owner);
        static int NativeStructural::testForAddedMethod (Object *owner);

        /* Entry points for the implementation of the reflection primitives. Usually only a
         * call to the associated method from the above group is performed, protecting some
         * memory from GC first.*/
        static FCDECL2(Object *, NativeStructural::getField, Object *owner, StringObject
                                     *name);

        static FCDECL2(Object *, NativeStructural::getFieldValue, Object *owner, StringObject *
                                     name);

        static FCDECL2(void, NativeStructural::addField, Object * owner,
                                     RuntimeStructuralFieldInfo *rsfi);

        static FCDECL2(void, NativeStructural::maskField, Object *owner,
                                     RuntimeStructuralFieldInfo *field);

        static FCDECL3(BOOL, NativeStructural::auxRemoveField, Object *owner, StringObject
                                     *name, BOOL overrideDeleted);

        static FCDECL2(void, NativeStructural::removeField, Object *owner, StringObject *name);

        /*Change a field charasteristics using the new ones provided in the structure. It's
```

```
        assumed that the structure's field name is the one who is going to be changed.*/
        static FCDECL2(void, NativeStructural::alterField, Object *owner,
                                    RuntimeStructuralFieldInfo *rsfi);

        //Change a field value (not the field itself!).
        static FCDECL2(void, NativeStructural::setFieldValue, RuntimeStructuralFieldInfo
                                    *field, Object *val);

        static FCDECL2(BOOL, NativeStructural::existField, Object *owner, StringObject *name);

        /************************* METHOD REFLECTION ****************************/

        /* Auxiliar functions. These functions are needed because we cannot invoke directly the
         * FCDELCX functions from the rest of VM code, only a caller placed into the managed
         * part is allowed to do so.*/
        static char * NativeStructural::convertNativeSig(char *sig, char *name, int mode);

        static OBJECTREF NativeStructural::auxExistMethod (Object *owner, StringObject
                                *signature, int *isClass, int *isAdded, int *addedDelControl);

        static Object * NativeStructural::auxGetMethod(Object *owner, StringObject *sig, int
                                *isClass, int *isAdded, int *addedDelControl);

        static BOOL NativeStructural::internalAuxRemoveMethod(Object *owner, StringObject
                                *name, BOOL overrideDeleted);

        static OBJECTREF NativeStructural::existAddedMethod (Object *owner, StringObject
                                *name);

        static void NativeStructural::internalAddMethod(Object *owner, MethodWrap *mw, BOOL
                                existTest);

        /* Entry points for the implementation of the reflection primitives. Usually only a
         * call to the associated method from the above group is performed, protecting some
         * memory from GC first.*/
        static FCDECL2(void, NativeStructural::maskMethod, Object *owner, MethodWrap *met);

        static FCDECL2(Object *, NativeStructural::internalGetMethod, Object *owner,
                                                                StringObject *signature);

        static FCDECL2(void, NativeStructural::addMethod, Object *owner, MethodWrap *miP);

        static FCDECL3(BOOL, NativeStructural::auxRemoveMethod, Object *owner, StringObject
                                                        *name, BOOL overrideDeleted);

        static FCDECL3(void, NativeStructural::alterMethod, Object *owner, StringObject *sig,
                                                        Object *val);

        static FCDECL2(BOOL, NativeStructural::existMethod, Object *owner, StringObject *sig);
};
```

20 Apéndice C: Tablas de Datos

Éstas son las tablas de datos de los gráficos del capítulo de mediciones.

Primitiva	ActivePython	CPython	Jython	IronPython	ЯRotor
Añadir atributos int a un objeto	590	541	20679	102327	75
Añadir atributos object a un objeto	611	580	2029	108095	75
Añadir atributos int a una clase	551	591	20063	104169	75
Añadir atributos object a una clase	661	610	2032	107184	75
Borrar atributos int de un objeto	561	591	18406	296366	150
Borrar atributos object de un objeto	611	601	19028	295344	150
Borrar atributos int de una clase	540	561	18536	300171	150
Borrar atributos object de una clase	581	560	18896	304758	150
Acceder a atributos añadidos a un objeto	521	530	18607	293782	75
Acceder a atributos inexistentes de un objeto	641	601	20019	105451	506
Acceder a atributos añadidos a una clase	511	481	18577	297327	75
Acceder a atributos inexistentes de una clase	611	571	20028	105241	525
Añadir métodos a un objeto	171	181	5859	9500	169
Añadir métodos a una clase	160	160	5839	9600	169
Invocar métodos añadidos a un objeto	160	170	5087	21050	75
Invocar métodos inexistentes de un objeto	190	210	6339	10400	131
Invocar métodos añadidos a una clase	181	180	5238	21430	75
Borrar métodos añadidos a un objeto	131	171	4627	19728	38
Borrar métodos añadidos a una clase	160	140	4697	20018	38

Tabla 1. Rendimiento de las distribuciones de *Python* y *ЯRotor* (ms)

Primitiva	ActivePython	CPython	Jython	IronPython	ЯRotor
Añadir atributos int a un objeto	4024	4040	18116	109116	8472
Añadir atributos object a un objeto	5876	5880	21492	110912	7768
Añadir atributos int a una clase	4404	4392	18556	123688	8472
Añadir atributos object a una clase	6256	6252	21892	124828	7768
Borrar atributos int de un objeto	4012	4004	18856	170068	9124
Borrar atributos object de un objeto	5872	5864	21900	176132	8548
Borrar atributos int de una clase	4392	4384	19196	168948	9124
Borrar atributos object de una clase	6252	6244	21968	177404	8556
Acceder a atributos añadidos a un objeto	4016	4004	18564	175972	9224
Acceder a atributos inexistentes de un objeto	4440	4424	18396	200000	9224
Acceder a atributos añadidos a una clase	3512	3504	16892	109656	6640
Acceder a atributos inexistentes de una clase	3512	3504	16540	109352	6640
Añadir métodos a un objeto	3428	3428	16620	45464	36572
Añadir métodos a una clase	3564	3560	16672	43948	36572
Invocar métodos añadidos a un objeto	3432	3432	16784	65480	42732
Invocar métodos inexistentes de un objeto	3440	3436	16972	46268	6224
Invocar métodos añadidos a una clase	3564	3564	16932	65228	42732
Borrar métodos añadidos a un objeto	3432	3432	16696	64368	42032
Borrar métodos añadidos a una clase	3568	3560	16440	64148	42036

Tabla 2. Consumo de memoria de las distribuciones de *Python* y *ЯRotor* (Kb)

Primitiva	ActivePython	CPython	Jython	IronPython
Añadir atributos int a un objeto	6,87	6,21	274,72	1363,36
Añadir atributos object a un objeto	7,15	6,73	26,05	1440,27
Añadir atributos int a una clase	6,35	6,88	266,51	1387,92
Añadir atributos object a una clase	7,81	7,13	26,09	1428,12
Borrar atributos int de un objeto	2,74	2,94	121,71	1974,77
Borrar atributos object de un objeto	3,07	3,01	125,85	1967,96
Borrar atributos int de una clase	2,60	2,74	122,57	2000,14
Borrar atributos object de una clase	2,87	2,73	124,97	2030,72
Acceder a atributos añadidos a un objeto	5,95	6,07	247,09	3916,09
Acceder a atributos inexistentes de un objeto	0,27	0,19	38,56	207,40
Acceder a atributos añadidos a una clase	5,81	5,41	246,69	3963,36
Acceder a atributos inexistentes de una clase	0,16	0,09	37,15	199,46
Añadir métodos a un objeto	0,01	0,07	33,67	55,21
Añadir métodos a una clase	-0,05	-0,05	33,55	55,80
Invocar métodos añadidos a un objeto	1,13	1,27	66,83	279,67
Invocar métodos inexistentes de un objeto	0,45	0,60	47,39	78,39
Invocar métodos añadidos a una clase	1,41	1,40	68,84	284,73
Borrar métodos añadidos a un objeto	2,45	3,50	120,76	518,16
Borrar métodos añadidos a una clase	3,21	2,68	122,61	525,79
PROMEDIO	**3,17**	**3,14**	**113,24**	**1246,18**

Tabla 3. Nº de veces en el que una primitiva es más lenta que la equivalente en *ЯRotor*

Primitiva	ActivePython	CPython	Jython	IronPython
Añadir atributos int a un objeto	1,11	1,10	-0,53	11,88
Añadir atributos object a un objeto	0,32	0,32	-0,64	13,28
Añadir atributos int a una clase	0,92	0,93	-0,54	13,60
Añadir atributos object a una clase	0,24	0,24	-0,65	15,07
Borrar atributos int de un objeto	1,27	1,28	-0,52	17,64
Borrar atributos object de un objeto	0,46	0,46	-0,61	19,61
Borrar atributos int de una clase	1,08	1,08	-0,52	17,52
Borrar atributos object de una clase	0,37	0,37	-0,61	19,73
Acceder a atributos añadidos a un objeto	1,30	1,30	-0,50	18,08
Acceder a atributos inexistentes de un objeto	1,08	1,08	-0,50	20,68
Acceder a atributos añadidos a una clase	0,89	0,89	-0,61	15,51
Acceder a atributos inexistentes de una clase	0,89	0,89	-0,60	15,47
Añadir métodos a un objeto	9,67	9,67	1,20	0,24
Añadir métodos a una clase	9,26	9,27	1,19	0,20
Invocar métodos añadidos a un objeto	11,45	11,45	1,55	0,53
Invocar métodos inexistentes de un objeto	0,81	0,81	-0,63	6,43
Invocar métodos añadidos a una clase	10,99	10,99	1,52	0,53
Borrar métodos añadidos a un objeto	11,25	11,25	1,52	0,53
Borrar métodos añadidos a una clase	10,78	10,81	1,56	0,53
PROMEDIO	**3,90**	**3,91**	**0,06**	**10,90**

Tabla 4. Uso de memoria global comparado de *ЯRotor* frente a las distintas distribuciones de *Python*

Primitiva Tommti	ЯRotor	CPython	ЯRotor/CPython
Int arithmetic	1800	27169	6,2%
Double arithmetic	6900	46196	13,0%
long arithmetic	4537,2	64483	6,6%
Trig	543,6	6960	7,2%
IO	561,6	110	83,6%
Array	411,6	7401	5,3%
Exception	2211,6	1061	67,6%
Hashmap	93,6	70	57,2%
Hashmaps	1105,2	241	82,1%
Heapsort	318	2493	11,3%
Vector	150	80	65,2%
Matrix multiply	19537,2	163315	10,7%
Nested Loop	1761,6	41450	4,1%
String concat	1405,2	12758	9,9%

Tabla 5. Rendimiento comparado (ms), ante primitivas no reflectivas, de *ЯRotor* y *CPython*

Primitiva Tommti	ЯRotor	CPython	ЯRotor/CPython
Int arithmetic	4600	3804	54,74%
Double arithmetic	4616	3808	54,80%
long arithmetic	4600	3812	54,68%
Trig	4628	3820	54,78%
IO	5824	3820	60,39%
Array	4676	3980	54,02%
Exception	5420	3804	58,76%
Hashmap	5728	3956	59,15%
Hashmaps	5508	3884	58,65%
Heapsort	5392	5048	51,65%
Vector	5072	3812	57,09%
Matrix multiply	4604	4000	53,51%
Nested Loop	4592	3808	54,67%
String concat	20260	15788	56,20%

Tabla 6. Uso de memoria (Kb) comparado, ante primitivas no reflectivas, de *ЯRotor* y *CPython*

Primitiva Tommti	Veces más memoria	Veces más rápido
Int arithmetic	-0,17	14,09
Double arithmetic	-0,18	5,70
long arithmetic	-0,17	13,21
Trig	-0,17	11,80
IO	-0,34	-0,80
Array	-0,15	16,98
Exception	-0,30	-0,52
Hashmap	-0,31	-0,25
Hashmaps	-0,29	-0,78
Heapsort	-0,06	6,84
Vector	-0,25	-0,47
Matrix multiply	-0,13	7,36

Nested Loop	-0,17	22,53
String concat	-0,22	8,08

Tabla 7. Uso de memoria vs rendimiento ofrecido para cada primitiva (*CPython - ЯRotor*)

Primitiva Tommti	ЯRotor	SSCLI	% adicional de ЯRotor
Int arithmetic	4600	4260	7,98%
Double arithmetic	4616	4280	7,85%
long arithmetic	4600	4260	7,98%
Trig	4628	4284	8,03%
IO	5824	5364	8,58%
Array	4676	4340	7,74%
Exception	5420	5068	6,95%
Hashmap	5728	5384	6,39%
Hashmaps	5508	5168	6,58%
Heapsort	5392	5060	6,56%
Vector	5072	4736	7,09%
Matrix multiply	4604	4268	7,87%
Nested Loop	4592	4260	7,79%
String concat	20260	19924	1,69%
PROMEDIO			**7,08%**

Tabla 8. Uso de memoria (Kb) comparado, ante primitivas no reflectivas, de *ЯRotor* y *SSCLI*

Primitiva Tommti	ЯRotor	SSCLI	% adicional ЯRotor
Int arithmetic	1800	1800	50,00%
Double arithmetic	6900	6900	50,00%
long arithmetic	4537,2	4555,2	49,90%
Trig	543,6	524,4	50,90%
IO	561,6	261,6	68,22%
Array	411,6	411,6	50,00%
Exception	2211,6	2024,4	52,21%
Hashmap	93,6	37,2	71,56%
Hashmaps	1105,2	168	86,80%
Heapsort	318	337,2	48,53%
Vector	150	18	89,29%
Matrix multiply	19537,2	19630,8	49,88%
Nested Loop	1761,6	1761,6	50,00%
String concat	1405,2	300	82,41%

Tabla 9. Rendimiento (ms) comparado, ante primitivas no reflectivas, de *ЯRotor* y *SSCLI*

Benchmarks	ЯRotor	SSCLI
lcscbench	6,68	3,7
ahcbench	17,37	5,53

Tabla 10. Tiempo (ms) de la ejecución de los *benchmark*

Benchmarks	ЯRotor	SSCLI
Lcscbench	35576	35184
Ahcbench	5872	5436

Tabla 11. Memoria ocupada (Kb) por la ejecución de los *benchmark*

21 APÉNDICE D: BIBLIOGRAFÍA

[Abelson00] H. Abelson y alumnos. *Scheme. Revised Report on the Algorithmic Language Scheme*. R. Kelsey, W. Clinger y J. Rees (Editores) (2000).

[Abelson06] H. Abelson, P. Greenspun, L. Sandon. *Tcl for Web nerds*.

[Achour06] M. Achour, F. Betz, A. Dovgal y otros. *PHP Manual*.

[Achour06b] M. Achour, F. Betz, A. Dovgal y otros. *PHP Manual: Reflection*.

[ActivePython06] ActiveState: Dynamic Tools for Dynamic Languages. *ActivePython*.

[Adobe06] Adobe Systems Incorporated. *Adobe Acrobat*.

[Agesen97] O. Agesen. *Design and implementation of Pep, a Java just-in-time translator*. Theory and Practice of Object Systems (1997).

[Agesen00] O. Agesen, L. Bak, C. Chambers. *The SELF 4.1 Programmer's Reference Manual*. Sun Microsystems, Inc. and Stanford University (2000).

[Aho90] A. V. Aho. *Compiladores: Principios, Técnicas y Herramientas*. Addison-Wesley Iberoamericana (1990).

[Altman00] E. Altman, M. Gschwind, S. Sathaye y otros. *BOA: The architecture of a binary translation processor*. Technical Report RC 21665, IBM Research Division, Yorktown Heights, NY. (2000).

[Álvarez99] D. Á. Gutiérrez. *Persistencia Completa para un Sistema Operativo Orientado a Objetos usando una Máquina Abstracta con Arquitectura Reflectiva*. Tesis Doctoral. Departamento de Informática. Universidad de Oviedo (1999).

[Amdhal00] G. M. Amdahl, G. A. Blaauw, F. P. Brooks JR. *Architecture of the IBM System/360*. IBM J. Res. Develop. VOL. 44 NO. 1/2 (2000).

[Andersen98] A. Andersen. *A note on reflection in Python 1.5*. Distributed Multimedia Research Group Report. MPG-98-05, Lancaster University (Reino Unido) (1998).

[AndroMDA07] andromda.org. *Cutting Edge MDSD/MDA Toolkit*.

[AnsiBasic87] ANSI Standard X3.113-1987. *Programing Languages Full Basic*. American National Standards Institute (1987).

[AnsiFortran97] ANSI X3.198-1992 (R1997). *American National Standard – Programming Language Fortran Extended*. American National Standards Institute (1997).

[Anthony05] D. Anthony, M. Leung, W. Srisa-an. *To JIT or not to JIT: The Effect of Code Pitching on the Performance of .NET Framework*. UNION Agency - Science Press, Plzen, Czech Republic (2005).

[AORTA06] Technische Universitat Darmstadt. *AORTA (Aspect-Oriented Run-Time Architecture)*.

[AORuby06] *Ruby Garden, AspectOrientedRuby.*

[Apache06] The Apache *Software* Foundation*. Apache Web Server.*

[Apple98] Apple Computer, Inc. *The Dylan Reference Manual* (1998).

[ArcStyler07] interactive-objects.com. *ArcStyler for IBM RSM Edition.*

[Armstrong97] J. Armstrong. *The development of Erlang*. Proceedings of ICFP '97 (1997).

[AspectProgramming06] Xerox Corporation. *The AspectJTM Programming Guide.*

[AspectProgramming06b] Eclipse.org. *Static crosscutting: Appendix B. Language Semantics.*

[AspectJ06] Eclipse.org. *AspectJ: Crosscutting objects for better modularity*.

[AspectWerkz06] *AspectWerkz - Plain Java AOP*.

[Assumpcao93] J. Assumpcao Jr. *O Sistema Orientado a Objetos Merlin em Máquinas Paralelas*. Anais do V SBAC-PAD. Florianópolis (Brasil) (1993).

[Assumpcao95] J. Assumpcao Jr., S. Takeo Kufuji. *Bootstrapping the Object-Oriented Operating System Merlin: Just add Reflection*. Meta'95 Workshop on Advances in Metaobject Protocols and Reflection. ECOOP (1995).

[Auslander96] J. Auslander, M. Philipose, C. Chambers y otros. *Fast, effective dynamic compilation*. Proceedings of PLDI'96. 149–159 (1996).

[Autonomic06] International Bussines Machines. *Autonomic Computing Homepage*.

[Aycock03] J. Aycock. *A Brief History of Just-In-Time*. ACM Computing Surveys (CSUR). Volume 35 , Issue 2 (2003).

[Ayers06] M. B. Ayers. *Advanced Access Content System (AACS): A Status Update Presented to CPTWG* (2006).

[Azevedo99] A. Azevedo, A. Nicolau, J. Hummel. *Java annotation-aware just-in-time (AJIT) compilation system*. Proceedings of *JAVA* '99. 142–151 (1999)

[Baillarguet98] C. Baillarguet, I. Piumarta. *An Highly-Configurable, Modular System Architecture for Mobility, Interoperability, Specilisation and Reuse*. INRIA, Rocquencourt, B.P. 105, 78153, Les Chesnay Cedex, Francia (1998).

[Baker97] S. Baker. *CORBA Distributed Objects*. Addison-Wesley, ACM Press. ISBN 0-201-92475-7 (1997).

[Baker02] J. Baker, W. Hsieh. *Runtime aspect weaving through metaprogramming*. AOSD 2002 conference Proceedings (2002)

[Bala99] V. Bala, E. Duesterwald, S. Banerjia. *Transparent dynamic optimization*. Technical Report. HPL-1999-77, Hewlett-Packard, Polo Alto, CA (1999).

[Ballard95] S. Ballard. *Comparison of Dylan to C++* (1995).

[Bancilhon92] F. Bancilhon, C. Belobel, P. Kanellakis. *Building an Object-Oriented Database System – The store of O2*. Morgan Kaufman (1992).

[Barendregt81] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam (1981).

[Beners96] T. Beners-Lee, R. Fielding, H. Frystyk. *Hypertext Transfer Protocol– HTTP 1.0*. HTTP Working Group (1996).

[Bik99] A. J. C. Bik, M. Girkar, M. R. Haghighat. *Experiences with Java JIT optimization*. Innovative Architecture for Future Generation High-Performance Processors and Systems. IEEE Computer Society Press, Los Alamitos, CA, 87–94 (1999).

[Binstock07] A. Binstock. *MDA gives us reason to believe in methodology ... at last*. devx.com.

[BitTorrent06] Bittorrent.com. *Sitio oficial de BitTorrent.*

[Blair97] G. S. Blair, G. Coulson. *The Case for Reflective Middleware*. Proceedings of the 3rd Cabernet Plenarg Workshop. Rennes (Francia) (1997).

[Blender06] Blender.org. *Blender: Open source 3D graphics creation*.

[Boehm06] H. Boehm. *A garbage collector for C and C++*.

[Böllert99] K. Böllert. *On weaving aspects*. European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming (1999).

[Boo05] CodeHaus.org**.** *Boo. A wrist friendly language for the CLI.*

[Boo05b] Codehaus.org. *Gotchas for Python Users*.

[Booch94] G. Booch. *Análisis y diseño orientado a objetos con aplicaciones*. Editorial Addison-Wesley / Díaz de Santos (1994).

[Borland06] Borland International. *Borland Worldwide.*

[Borning86] A. H. Borning. *Classes Versus Prototypes in Object-Oriented Languages*. Proceedings of the ACM/IEEE Fall Joint Computer Conference 36-40 (1986).

[Bracha93] G. Bracha, D. Griswold. *Strongtalk: Typechecking Smalltalk in a production environment*. Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, pages 215–230 (1993).

[Bradley06] B. L. Jones. *What Are Partial Types in C#?*.

[Brodie87] L. Brodie. *Starting Forth: an Introduction to the Forth language and Operating System for Beginners and Professionals*. Prentice Hall (1987).

[Bruckschlegel05] T. Bruckschlegel. *Microbenchmarking C++, C#, and Java*. Dr. Dobb's Portal.

[Bühler06] E. R. Bühler. *Optimizando aplicaciones de .NET Framework para la nueva tecnología HyperThreading*. MSDN Latinoamerica. VBLibros.com

[Burger97] R. G. Burger. *Efficient compilation and profile-driven dynamic recompilation in scheme*. Ph.D. dissertation, Indiana University, Bloomington, IN (1997).

[Burke99] M.G. Burke, J. D. Choi, S. Fink y otros. *The Jalapeño dynamic optimizing compiler for Java*. In Proceedings of *JAVA* '99. 129–141 (1999).

[Burton02] K. R. Burton, K. Burton. *.NET Common Language Runtime Unleashed*. ISBN: 0672321246 (2002).

[Cairo05] *Cairo Graphics Engine Homepage*.

[Campbell83] F. Campbell. *The Portable UCSD p-System*. Microprocessors and Microsystems, No. 7. pp. 394-398 (1983).

[Campione99] M. Campione, K. Walrath. *The Java Tutorial. Second Edition. Object Oriented Programming for Internet*. The *Java* Series. Sun Microsystems (1999).

[Cardelli96] L. Cardelli. *Object-based vs. Class-based Languages*. Digital Equipment Corporation Systems Research Center. PLDI'96 Tutorial (1996).

[Cardelli97] L. Cardelli. *Type Systems*. Handbook of Computer Science and Engineering, Chapter 103. CRC Press (1997).

[Cardelli06] L. Cardelli. *A Language with Distributed Scope*. Digital Equipment Corporation, Systems Research Center.

[CDRInfo06] Cdrinfo.com. *BDA Details Blu-Ray Disc Content Protection Technology*.

[Chambers89] C. Chambers., D. Ungar, E. Lee. *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. OOPSLA'89 Conference Proceedings. Published as SIGPLAN Notices, 24, 10, 49-70 (1989).

[Chambers90] C. Chambers, D. Ungar. *Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs*. Proceedings of PLDI '90 (1990).

[Chambers91] C. Chambers, D. Ungar. *Making Pure Object-Oriented Languages Practical*. Proceedings of OOPSLA '91 Conference, Phoenix, AZ. (1991).

[Chambers93] C. Chambers. *The Cecil Language: Specification and Rationale*. Technical Report TR-93-03-05. Department of Computer Science and Engineering. University of Washington (1993).

[Chen2000] W. K. Chen, S. Lerner, R. Chaiken, D. M. Gillies. *Mojo: a dynamic optimization system*. Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization (2000).

[Chiba95] S. Chiba. *A Metaobject Protocol for C++*. 10 th Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA'95 (1995).

[Chiba98] S. Chiba, M. Tatsubori. *Yet Another Java.lang.Class*. ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems (1998).

[Chiba06] S. Chiba. J*avassis*t

[Chiba06b] S. Chiba. *Javassist Tutorial.*

[Clark83] K. L. Clark, S. Gregory. *Parlog: A Parallel Logic Programming Language*. Imperial College, London (1983).

[CLSB06] *The Computer Language Shootout Benchmarks*.

[Cmelik94] B. Cmelik, D. Keppel. *Shade: A fast instruction-set simulator for execution profiling*. Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems (1994).

[Cocke70] J. Cocke. *Global Common Subexpression Elimination*. Proceedings of a Symposium on Compiler Construction, ACM SIGPLAN Notices (1970).

[Cocoa06] CocoaSharp.org. *CocoaSharp.org: Bringing C# and Mono to the Mac*.

[CodeGuru05] Code Guru. *Late Binding and On-the-Fly Code Generation Using Reflection in C#*.

[Codehaus06] Codehaus.org. *Boo: A wrist friendly language for the CLI: Closures*.

[Cohen04] T. Cohen, J. Gil. *AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework*. ECOOP 2004 – Object-Oriented Programming, 3086 pages 219-243, Springer-Verlag.

[Cointe88] P. Cointe. *The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System*. Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors. North-Holland (1988).

[Cointe92] P. Cointe, J. Malenfant, C. Dony, P. Mulet. *Etude de la réflexion de comportement dans le langage Self*. Premières Journées Représentation par Objects. La Grande Motte (Francia). (1992).

[Colwel88] R. P. Colwel, E. F. Gehringer, E. D. Jensen. *Performance Effects of Architectural Complexity in the Intel 432*. ACM Transactions on Computer Systems, Vol. 6, No. 3 (1988).

[Computer06] Computer and Information Services. *Job Control Language User's Manual*.

[Consel96] C. Consel, F. Noël. *General approach for run-time specialization and its application to C.* Proceedings of POPL '96 (1996).

[Consel98] C. Consel, L. Hornof, R. Marlet y otros. *Tempo: Specializing systems applications and beyond*. ACM Computation Surveys (1998).

[Cooper99] K. D. Cooper, P. J. Schielke, D. Subramanian. *Optimizing for reduced code space using genetic algorithms*. Proceedings of LCTES 1999 (1999).

[Cooper01] K. D. Cooper, D. Subramanian, L. Torczon. *Adaptive optimizing compilers for the 21st century*. Proceedings of the 2001 LACSI Symposium. Los Alamos Computer Science Institute (2001).

[Cooper05] K. D. Cooper, A. Grosul, T. J. Harvey y otros. *ACME: Adaptive Compilation Made Efficient*.

[Cooper05b] K. Cooper, A. Dasgupta, J. Eckhardt. *Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms*. Department of Computer Science, Rice University (2005).

[Cooper06] M. Cooper. *Advanced Bash-Scripting Guide An in-depth exploration of the art of shell scripting*.

[Cpan06] Cpan.org. *Perl Objects*.

[Cpan06b] Cpan.org. *Fast prototype-based OO programming in Perl*.

[CpanParrot06] Cpan.org. *Parrot Subsystem Porting Introduction*.

[CProgrammingHuffman06] CProgramming.com. *Humffman Enconding*.

[CPython06] Python.org. *Python Developers Guide*.

[Cramer97] T. Cramer, R. Friedman, T. Miller y otros. *Compiling Java just in time*. IEEE Micro 17 (1997).

[Crane05] D. Crane, E. Pascarello, D. James. *Ajax in Action*. Manning Publications (2005).

[Cueva91] J. M. Cueva Lovelle. *Lenguajes, Gramáticas y Autómatas*. ISBN: 84-600-7871-X (1991).

[Cueva92b] J. M. Cueva Lovelle. *Tablas de Símbolos en Procesadores de Lenguajes*. Cuaderno Didáctico número 54. Departamento de Matemáticas. Universidad de Oviedo (1992).

[Cueva93] J. M. Cueva Lovelle. *Análisis Léxico en Procesadores de Lenguaje*. Cuaderno Didáctico número 48. Departamento de Matemáticas. Universidad de Oviedo (1993).

[Cueva94] J. M. Cueva Lovelle, P.A. García Fuente, B. López Pérez, C. Luengo Díez y M. Alonso Requejo. *Introducción a la Programación Estructurada y Orientada a Objetos con Pascal*. ISBN: 84-600-8646-1 (1994).

[Cueva95] J. M. Cueva Lovelle. *Análisis Sintáctico en Procesadores de Lenguaje*. Cuaderno Didáctico número 61. Departamento de Matemáticas. Universidad de Oviedo (1995).

[Cueva95b] J. M. Cueva Lovelle. *Análisis Semántico en Procesadores de Lenguaje*. Cuaderno Didáctico número 62. Departamento de Matemáticas. Universidad de Oviedo (1995).

[Cueva98] J. M. Cueva Lovelle. *Conceptos Básicos de Procesadores de Lenguaje*. Cuaderno Didáctico número 10. Editorial Servitec (1998).

[DAJ06] *DAJ: Demeter in AspectJ Homepage.*

[Dakin73] R. J. Dakin, P. C. Poole. *A mixed code approach*. The Computation Journal (1973).

[Davis06] R. Davis. *RubyGarden.org: Next generation Ruby === Rite*.

[Dawson73] J. L. Dawson. *Combining interpretive code with machine code*. The Computation Journal (1973).

[DeMichiel87] L. G. DeMichiel, R. P. Gabriel. *The Common Lisp Object System: An Overview*. European conference on object-oriented programming on ECOOP '87 (1987)

[Denning05] P. J. Denning. *The locality principle*. Communications of the ACM archive, Vol. 48 , Issue 7, pp 19 - 24. ACM Press New York, NY, USA (2005)

[Deutsch84] L. P. Deutsh, A. M. Schiffman. *Efficient implementation of the Smalltalk-80 system*. In Proceedings of POPL '84 (1984).

[Devarticles06] Devarticles.com. *JavaScript: The Power of Javascript. Basic types of data*.

[Dewdney88] A. K. Dewdney. *The Armchair Universe: An Exploration of Computer World*. W. H. Freeman, ISBN: 0-7167-1939-8 (1988).

[Dewdney90] A. K. Dewdney. *The Magic Machine: A Handbook on Computer Sorcery*. W. H. Freeman, ISBN: 0-7167-2125-2 (1990).

[Dijkstra76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall (1976).

[Django06] djangoproject.com. *Django: The Web framework for perfectionist with deadlines*.

[Doederlein03] O. Doederlein. *The Tale of Java Performance*. Journal of Object Technology, vol. 2, no. 5, pp. 17-40 (2003).

[DotGNU05] DotGnu.org. *DotGNU project - GNU Freedom for the Internet*.

[Douence99] R. Douence, M. Südholt. *The next 700 Reflective Object-Oriented Languages*. École des mines de Nantes. Dept. Informatique (Francia). Technical report no.: 99-1-INFO (1999).

[Douence01] R. Douence, O. Motelet, M. Südholt. *A formal definition of crosscuts*. Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS Springer Verlag, (2001).

[Douence02] R. Douence, M. Südholt. *A model and a tool for Event-Based AOP*. Technical report no. 02/11/INFO, École des Mines de Nantes, at LMO'03, 2nd edition (2002).

[Dybvig06] R. K. Dybvig, M. Leone, D. Wise. *The Dynamo Project.*

[Ebcioglu96] K. Ebcioglu, E. R. Altman. *DAISY: Dynamic compilation for 100% architectural compatibility*. Technical Report RC 20538. IBM Research Division, Yorktown Heights, NY (1996)

[Eckel00] B. Eckel. *Thinking in Java, second edition*. Prentice Hall. ISBN 0-13-027363-5 (2000).

[Eckel00b] B. Eckel. *Thinking in C++*, second edition, volume 1. Prentice Hall (2000).

[Eckel04] B. Eckel. *Static vs. Dynamic*. Mindview.net.

[Eclipse06] Eclipse.org. *Eclipse IDE.*

[ECMA02] Ecma International. *Standard ECMA 335 – Common Language Infraestructure (CLI), Second Edition*.

[ECMA06] Ecma International. *Ecma International Homepage.*

[ECMA26299] Ecma International. *Standard ECMA-262: ECMAScript Language Specification* (1999).

[ECMA33405] Ecma International. *Standard ECMA-334. C# Language Specification. 3rd Edition* (2005).

[Emerson06] The Cherry L. Emerson Center for Scientific Computation. *Perl Manual*.

[ETHOberon06] ETH Zurich. *ETH Oberon Homepage*.

[EventHelix06] EventHelix.com. *Optimizing C and C++ Code*.

[Evins94] M. Evins. *Objects Without Classes*. Computer IEEE. Vol. 27, N. 3, 104-109 (1994).

[Evins05] M. Evins. *MacTech. The Journal of Macintosh technology*.

[Feinberg97] N. Feinberg, S. E. Keene, R. Mathews, P. T. Withington. *Dylan Programming*. Addison-Wesley Longman Publishing Company Inc. (1997).

[Ferber88] J. Ferber. *Conceptual Reflection and Actor Languages*. Meta-Level Architectures and Reflection. P. Maes and D. Nardi Editors.North-Holland (1988).

[Ferg06] S. Ferg. *Python & Java: a Side-by-Side Comparison*.

[Ferreira98] L. L. Ferreira, C. M. F. Bubira. *The Reflective State Patern*. OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá) (1998).

[Fetting05] A. Fettig. *Twisted Network Programming Essentials.* O'Reilly (2005).

[Firaxis05] Firaxis Games. *Sid Meier's Civilization 4*.

[Folliot97] B. Folliot, I. Piumarta, F. Reccardi. *Virtual Virtual Machines*. Cabernet Radicals Workshop, Creta (1997).

[Folliot98] B. Folliot, I. Piumarta, F. Reccardi. *A Dynamically Configurable, Multi-Language Execution Platform*. 8th ACM SIGOPS European Workshop (1998).

[Foote90] B. Foote. *Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?* ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures (1990).

[Foote92] B. Foote. *Objects, Reflection, and Open Languages*. Workshop on Object-Oriented Reflection and Metalevel Architectures. ECOOP'92. Utrect (Holanda) (1992).

[Franz94] M. Franz. *Code-generation on-the-fly: A key to portable software*. Ph.D. dissertation. ETH Zurich, Zurich, Switzerland (1994).

[Freier96] A. O. Freier, P. Karlton, P. C. Kocker. *The SSL Protocol, version 3.0*. Transport Layer Security Working Group (1996).

[Gamma95] E. Gamma, R. Helm, J. Vlissides. *Design Patterns Applied*, Tutorial OOPSLA (1995).

[Gardens06] Queensland University Of Technology. *Gardens Point Ruby .NET Compiler Homepage*.

[Geist94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, V. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press. Cambridge, Massachusetts, EE.UU (1994).

[Gene00] G. Callahan. *Secrets of Java Serialization*.

[Gite06] V. G. Gite. *Linux Shell Scripting Tutorial v1.05r3. A Begginer's Handbook*.

[Glover04] A. Glover. *Feeling Groovy: Introducing a new standard language for the Java platform*. Ibm.com (2004).

[Glover05] A. Glover. *Practically Groovy: Of MOPs and mini-languages: Groovy moves the Meta Object Protocol out of the lab and into your apps*. Ibm.com (2005).

[GNU06] GNU.org. *GNU General Public License*.

[GNU06b] GNU.org. *GNU Lesser General Public License*.

[GOF94] E. Gamma, R. Helm, R. Johnson, J.O. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Editorial (1994).

[Goldberg83] A. Goldberg, D. Robson. *Smalltalk-80: The language and its Implementation*. Addison-Wesley (1983).

[Goldberg89] A. Goldberg, D. Robson. *Smalltalk-80: The language*. Addison-Wesley (1989).

[Golm97] M. Golm. *Design and Implementation of a Meta Architecture for Java*. Friedrich-Alexander-Universität. Computer Science Department. Erlangen-Nürnberg, Alemania (1997).

[Golm97b] M. Golm, J. Kleinöder. *Implementing Real-Time Actors with MetaJava*. ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems. Jyväskylä (Finlandia) (1997).

[Golm97c] M. Golm, J. Kleinöder. *MetaJava – A Platform for Adaptable Operating-System Mechanisms*. ECOOP'97 Workshop on Reflective Real-time Object-Oriented Programming and Systems. Jyväskylä (Finlandia) (1997).

[Golm98] M. Golm, J. Kleinöder. *metaXa and the Future of Reflection*. OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá) (1998).

[Gonzalez04] S. González, W. De Meuter, P. Costanza, S. Ducasse, R. Gabriel y T. DrsquoHondt. *2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity*. ECOOP 2004 Workshop Reader, Lect. Notes in C. Science. V. 3344 (2004)

[Gosling96] J. Gosling, B. Joy, G. Seele. *The Java Language Specification*. Addison-Wesley. (1996).

[Gowing96] B. Gowing, V. Cahill. *Meta-Object Protocols for C++: The Iguana Approach*. Distributed Systems Group, Department of Computer Science, Trinity College. Dublin (Irlanda). (1996).

[Grails06] codehaus.org. *GRAILS: See the light - agile, industrial strength, rapid web application development made easy*.

[Gregory87] S. Gregory. *Parallel Logic Programming in Parlog, The Language and its Implementation*. Addison-Wesley (1987).

[Groovy06] Groovy API Documentation. *Interface groovy.lang.GroovyObject*.

[Groovy06b] Groovy API Documentation. *class groovy.lang.Metaclass*.

[Gtk05] *Gtk#. The Free .NET GUI*.

[Gwydion05] Gwydion Dylan Homepage. *The Dylan Reference Manual*.

[Gwydion05b] OpenDylan.org. *The Open Dylan Project*.

[Gwydion05c] Gwydion Dylan Homepage. *The Dylan Reference Manual*: *The Dylan Introspection Module*.

[Hammond77] J. Hammond. *BASIC - an evaluation of processing methods and a study of some programs*. *Software* Practice and Experience (1977).

[Hansen74] G. J. Hansen. *Adaptive systems for the dynamic run-time optimization of programs*. Ph.D. dissertation. Carnegie-Mellon University, Pittsburgh, PA (1974).

[Hardwick97] J. Hardwick. *Java Microbenchmarks* (1997).

[Harris99] T. L. Harris. *An Extensible Virtual Machine Architecture*. Citrix Ssytems (Cambridge) Ltd., Computer Laboratory (1999).

[Haskell05] Haskell.org. *Haskell. A purely functional Language*.

[Hawaii01] University of Hawaii at Manoa College of Engineering. *The C Shell Tutorial*.

[Haygood94] R. C. Haygood. *Native code compilation in SICStus Prolog*. Proceedings of the Eleventh International Conference on Logic Programming. 190–204 (1994).

[HiPE06] Uppsala Universiter. *The High-Performance Erlang Project*.

[Holub03] A. Holub. *Why extends is evil*. Java Toolbox (2003)

[Hölzle94] U. Hölzle, D. Ungar. *A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance*. OOPSLA '94 Conference Proceedings, Portland, OR. Octubre (1994).

[Hölzle94b] U. Hölzle, D. Ungar. *Optimizing dynamically-dispatched calls with run-time type feedback*. Proceedings of PLDI '94 (1994).

[Hölzle95] U. Hölzle, D. Ungar. *Do Object-Oriented Languages Need Special Hardware Support?* ECOOP' 95 Conference Proceedings, Springer Verlag Lecture Notes on Computer Science 952. (1995).

[HotSwap06] Sun Developer Network. *White Parer: The Java HotSpot Performance Engine Architecture*.

[Howe99] D. Howe. "*Virtual Machine*". The Free Online Dictionary (1999).

[Huggins96] J. Huggins. "*Abstract State Machines*" (1996).

[Huggins99] J. Huggins. *Abstract State Machines: Introduction* (1999).

[Hürsch95] W.L. Hürsch, C.V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, (Northeastern University, Boston) (1995).

[IBM00] IBM Corporation. *VM/ESA – An S/390 Platform for Business Solutions*. VM/ESA Version 2 Release 4 Specification Sheet (2000).

[IBM00b] IBM Corporation. *The IBM J9 Virtual Machine* (2000).

[IBM00c] IBM Corporation. *Visual Age Micro Edition. Strategic Benefis of Virtual-Machine-Based Architecture. Core Technology for Embedded Systems* (2000).

[IBM00d] IBM Corporation. *Developing Embedded Applications* (2000).

[IBM06] International Bussines Machines. *IBM Jikes Research Virtual Machine homepage*.

[IBMResearch05] IBM Corporation. *Jalapeño Virtual Machine Project*.

[ICE05] Zeroc.com. *ZeroC. The Home of ICE*.

[IIS06] Microsoft Windows Server System. *Internet Information Services*.

[Igamberdiev04] M. Igamberdiev. *Analysis of available MDA-support tools*. Muzaffar Igamberdiev Company : GigSoft. Freeband A-MUSE project partners (2004).

[Ingalls78] D. H. Ingalls. *The Smalltalk-76 Programming System Design and Implementation* Conference Record on the 5 th Annual ACM Symposium on Principles of Programming Languages (1978).

[IronPython06] *IronPython Homepage*.

[Irvine99] *p-System: Description, Background, Utilities*. Educational Technology Center. Department of Information and Computer Science, University of California, Irvine (1999).

[Iverson62] K. E. Iverson. *A Programming Language* (1962).

[Izquierdo02] R. Izquierdo C. *RDM: Arquitectura Software para el Modelado de Dominios en Sistemas Informáticos.* Tesis Doctoral. Departamento de Informática. Universidad de Oviedo (2002).

[Jackson05] J. Jackson, K.-H. Tan. *The Architecture of Python*.

[Javascriptkit06] Javascriptkit.com. *Using the prototype object to add custom properties to objects*.

[JBoss06] JBoss.com. *JBoss. The Professional Open Source Company.*

[Jensen91] K. Jensen K., N. Wirth. *PASCAL User Manual and Report ISO Pascal Standard*. 4º Edición Springer-Verlag (1991).

[Jeswin06] P. Jeswin. *Xml Performance: XmlMark Revisited; Java, Mono and .NET*.

[JikesRVM05] *Jikes Virtual Machine*.

[Johanson00] E. Johansson, M. Pettersson, K. Sagonas. *A high performance Erlang system*. Proceedings of PPDP '00 (2000).

[Johnston00] S. J. Johnston. *The Future of COM+ – Microsoft's .NET Revealed*. The XML Magazine (2000).

[Jones99] P. Jones. *VMware Virtual Platform for Linux: Beyond Dual-Booting*. Internet.com (1999).

[JSR06] *JSR 223: Scripting for the JavaTM Platform*.

[JUnit06] Object Mentor. *Junit, Testting Resources for Extreme Programming*.

[Jython06] Jython.org. *Jython Homepage*.

[Kalev98] D. Kalev. *The ANSI/ISO C++ Professional Programmers Handbook*. Que Editorial (1998).

[Keleher96] P. Keleher. *CVM: The Coherent Virtual Machine*. University of Maryland (1996).

[Kiczales91] G. Kiczales, J. des Rivières, D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press (1991).

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda y otros. *Aspect Oriented Programming*. Proceedings of ECOOP'97 Conference. Finlandia (1997).

[Kirby92] G. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems*. Ph.D. Thesis, University of St Andrews (1992).

[Kirby98] G. Kirby, R. Morrison, D. Stemple. *Linguistic Reflection in Java*. *Software* Practice & Experience, 28 (1998).

[Kistler99] T. Kistler, M. Franz. *The case for dynamic optimization: Improving memoryhierarchy performance by continuously adapting the internal storage layout of heap objects at run-time*. Technical Reports. University of California, Irvine, Irvine, CA (1999).

[Klaiber00] A. Klaiber. *The technology behind Crusoe processors*. Technical Reports (Jan.), Transmeta Corporation, Santa Clara, CA (2000).

[Kleinöder96] J. Kleinöder, M. Golm. *MetaJava: An Efficient RunTime Meta Architecture for Java™*. Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS'96). Seattle, Washington (EE.UU.) (1996).

[Kloosterman98] S. Kloosterman, M. Shapiro. *Large Scale Distributed Object Programming Made Easy* (1998).

[Knuth71] D. E. Knuth. *An empirical study of Fortran programs*. *Software* Practice and Experience (1971).

[Koelbel94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele y M. E. Zosel. *The High Performance Fortran Handbook*. Editorial Zosel (1994).

[Kozak00] R. Kozak. *The Dish on Kylix. Cross-Platform Controls. From Windows to Linux, and Back* (2000).

[Krall97] A. Krall, R. Grafl. *A Java just-in-time compiler that transcends JavaVM's 32 bit barrier*. Proceedings of PPoPP '97 Workshop on *Java* for Science and Engineering (1997).

[Kramer96] D. Kramer. *The Java Platform. A White Paper*. Sun MicroSystems JavaSoft (1996).

[Krasner83] G. Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley (1983).

[Kulkarni03] P. Kulkarni, W. Zhao, H. Moon y otros. *Finding effective optimization phase sequences*. Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Tools, and Compilers for Embedded Systems (2003).

[Kumar06] G. Kumar. *A JIT (Just In Time) Compiler for Microsoft .NET CLR (Common Language Runtime) on IA-64*.

[Laird02] C. Laird, K. Soraiz. *Regular Expressions: Syntax Checking the Scripting Way* (2002).

[Lam02] iunknown.com. *Letter is Better: John Lam on software*.

[Leavenworth93] B. Leavenworth. *PROXY: a Scheme-Based Prototyping Language*. Dr. Dobb's Journal, No. 198, 86-90 (1993).

[Ledoux96] T. Ledoux, P. Cointe. *Explicit Metaclasses as a Tool for improving the Design of Class Libraries*. Proceedings of ISOTAS'96, Springer-Verlang, Kanazawa (Japón) (1996).

[Ledoux99] T. Ledoux. *OpenCorba: a Reflective Open Broker*. Lecture Notes in Computer Science, vol. 1616 (1999).

[Leone94] M. Leone, P. Lee. *Lightweight run-time code generation*. Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (1994).

[Levine03] N. Levine. *Fundamentals of CLOS*. Ravenbrook Limited (2003).

[Li89] K. Li, P. Hudak. *Memory coherence in shared virtual memory systems*. ACM Transactions on Computer Systems, 7 (1989).

[Lieberman86] H. Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. OOPSLA'86 Conference Proceedings. Published as SIGPLAN Notices, 21, 11, 214-223 (1986).

[Lindholm96] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems. (1996).

[LispWorks06] LispWorks. *The Common Lisp Standard*.

[Lua06] Lua.org. *Lua: The Programming Language*.

[Lynagh03] I. Lynagh. *Template Haskell* (2003).

[Macrakis93] S. Macrakis. *Delivering Applications to Multiple Platforms Using ANDF*. AIXpert (1993).

[Maes87] P. Maes. *Computational Reflection*. PhD. Thesis. Laboratory for Artificial Intelligence, Vrije Universiteit Brussel. Bruselas, Bélgica (1987).

[Maes87b] P. Maes. *Concepts and experiments in computational reflection*. Proceedings of OOPSLA'87, pages 147--155. Orlando (1997).

[Mandado73] E. Mandado. *Sistemas Electrónicos Digitales*. Marcombo Boixareu Editores (1973).

[Manzoor06] K. Manzoor. *The Common Language Runtime (CLR) and Java Runtime Environment (JRE)*.

[Marlow05] S. Marlow. *GHC Homepage*.

[Matsuoka91] S. Matsuoka, T. Watanabe, A. Yonezawa. *Hubrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91), Ginebra (Suiza) (1991).

[Matthijs97] F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, P. Verbaten. *Aspects should not die*. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect-Oriented Programming (1997).

[May87] C. May. *Mimic: A fast System/370 simulator*. Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques (June). ACM Press (1987).

[McCallister04] B. McCallister, G. Laforge. *Groovy Mixin Proposal*. CodeHaus.org (2004).

[McClean06] J. McClean. *Painless AOP with Groovy*. Infoq.com.

[MediaWiki06] MediaWiki.org. *MediaWiki*.

[Megginson00] D. Megginson. *SAX 2.0: The Simple API for XML*.

[Meier05] E. Meijer, P. Drayton. *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*. Microsoft Corporation.

[Mendhekar92] A. Mendhekar, D. P. Friedman. *Towards a Theory of Reflective Programming Languages*. Department of Computer Science. Indiana University (EE.UU.) (1992).

[Mevel87] A. Mével, T. Guéguen. *Smalltalk-80*. Mac Millan Education, Houndmills, Basingstoke. (1987).

[Mezini03] M. Mezini, K.Ostermann. *Conquering Aspects with Caesar*. In (M. Aksit ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA. ACM Press (2003).

[Microsoft05] Microsoft Corporation. *Microsoft .NET Homepage.*

[Microsoft06] Microsoft Developer Network. *NET Framework Class Library. System.Reflection Namespace.*

[MicrosoftCOFF06] Microsoft Help and Support. *Common Object File Format (COFF)*.

[MicrosoftSSCLI06] Microsoft.com. *Shared Source Common Language Infrastructure 1.0 Relase*.

[MicrosoftSSCLI06b] Microsoft.com. *Shared Source Common Language Infrastructure 2.0 Release*.

[Mitchell68] J. G. Mitchell, A. J. Perlis, H. R. Van Zoeren. *LC2: A language for conversational computing*. Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds, Eds. Academic Press, New York, NY. (Proceedings of 1967 ACM Symposium) (1968).

[Mock99] M. Mock, M. Berryman, C. Chambers, S. J. Eggers. *Calpa: A tool for automating dynamic compilation*. Proceedings of the Second ACM Workshop on Feedback-Directed and Dynamic Optimization (1999).

[Mono05] MonoProject.com. *What is Mono exactly?*.

[Mono06] MonoProject.com. *The Mono Project*.

[MonoRoad05] MonoProject.com. *The Mono Project Roadmap*.

[MonoVSDotGNU05] MonoProject.com. *Mono and Portable .NET*.

[Mosses92] P. D. Mosses. *Action Semantics*. Cambridge University Press (1992).

[MozillaJavaS06] Mozilla.org. *JavaScript*.

[MSDNCLR06] Microsoft Developer Network. *.NET Framework Developer Center: The Common Language Runtime (CLR)*.

[MSDNComp06] Microsoft Developer Network. *Programar con componentes*.

[MSDNEvents06] Microsoft Developer Network. *Events Tutorial*.

[MSDNJS06] Microsoft Developer Network. *JScript .NET*.

[MSDNNET3006] Microsoft Developer Network. *Learn .NET Framework 3.0*.

[MSDNNET3006b] Microsoft Developer Network. *Manual del programador de .NET Framework: Información general acerca de .NET Framework*.

[MSDNPInvoke06] Microsoft Developer Network. *.NET Framework Developer Center: Interoperability*.

[MSDNRemoting06] Microsoft Developer Network. *.NET Remoting Overview: .NET Remoting Architecture*.

[MSDNSSCLI06] Microsoft Developer Network. *Rotor: shared Source CLI Provides Source Code for a FreeBSD Implementation o .NET*.

[MSDNTypes06] Microsoft Developer Network. *.NET Framework Developer's Guide: Common Type System Overview.*

[MSDNVB06] Microsoft Developer Network. *VBScript: Visual Basic Scripting Edition*.

[MSRPhoenix06] Microsoft Research. *Phoenix Framework*.

[MSRRFP06] Microsoft Research. *Rotor Projects*.

[MSRRFP06b] Microsoft Research. *Rotor funded projects in 2004*.

[Mukerji01] J. Mukerji, J. Miller. *The MDA Guide: Overview and guide to OMG's architecture*. omg.org.

[Mulet93] P. Mulet, P. Cointe. *Definition of a Reflective Kernel for a Prototype-Based Language.* International Symposium on Object Technologies for Advanced *Software*. Kanazawa (Japón) (1993).

[Mulet94] P. Mulet, T. Ledoux, D. Barbaron, F. Rivard, P. Cointe. *Importing SOM Libraries into Classtalk*. OOPSLA'94 Workshop on Experiences with CORBA: Is CORBA ready for duty? (1994).

[Mulet95] P. Mulet, J. Malenfant, P. Cointe. *Towards a Methodology for Explicit Composition of MetaObjects*. OOPSLA'95 Conference Proceedings, ACM Sigplan Notices (1995).

[Murata94] K. Murata, R. N. Horspool, Y. Yokote, E. G. Mannig, M. Tokoro. *Cognac: a Reflective Object-Oriented Programming System using Dynamic Compilation Techniques*. Proceedings of the annual Conference of Japan Society of *Software* Science and Technology (JSSS'94) (1994).

[MySQL06] MySQL.com. *MySQL: The world's most popular open source database*.

[Neumann03] M. Neumann. *How Ruby Sucks* (2003).

[Ng76] T. S. NG, A. Cantoni. *A Runtime interaction with FORTRAN using mixed code*. The Computation Journal (1976).

[Noras04] A. Noras. *Declarative JavaScript programming*.

[Nori76] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli y C. Jacobi. *The Pascal-P Compiler: Implementation Notes*. Bericht 10, Eidgengössische Technische Hochschule. Zurich, Suiza (1976).

[Notario06] D. Notario. *The CLR x86 JIT, an overview*.

[Novell06] Novell.com. *Novell: Software for the Open Enterprise*.

[NSA06] National Security Agency - Central Security Service. *Technology Profile Fact Sheet. Title: NetTop.*

[Nullstone06] Nullstone. *"NULLStone Optimization Categories"*.

[Nunit05] Nunit.org. *Nunit Homepage*.

[Oliva98] A. Oliva, L. E. Buzato. *An overview of MOLDS: A Meta-Object Library for Distributed Systems*. Segundo Workshop em Sistemas Distribuidos, Curitiba (Brasil) (1998).

[Oliva98b] A. Oliva, I. Calciolari G., L. E. Buzato. *The reflexive architecture of Guanará*. Technical Report IC-98-14, Instituto de Compurtaçao, Universidad de Campinas (1998).

[Oliva98c] Alexandre Oliva, Luiz Eduardo Buzato. *The implementation of Guanará on Java*. Technical Report IC-98-32, Instituto de Compurtaçao, Universidad de Campinas. Septiembre de 1998.

[Oliva99] A. Oliva, L. E. Buzato. *The Design and Implementation of Guanará*. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99). San Diego (EE.UU.) (1999).

[OMG95] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification* (1995).

[OMG98] Object Management Group (OMG). *CORBA Components. CORBA 3.0 Draft Specification* (1998).

[OMG07] OMG.org. *The Object Management Group (OMG) Homepage*.

[OMGMDA07] OMG.org. *MDA: The Architecture of Choice for a Changing World®: Model Driven Architecture (MDA) FAQ.*

[OMGQVT07] OMG.org. *Catalogue of OMG Modelling and Metadata Specifications: MOF™ Query / Views / Transformations*.

[OpenSource06] Open Source TM. *The MIT License*.

[OptimalJ07] compuware.com. *OptimalJ: Model-driven development for Java*.

[Oracle06] Oracle.com. *Oracle Corporation*.

[OReilly06] O'Reilly Media. *Perl.com. The Source for Perl*.

[OReilly06b] O'Reilly Media. *Advanced Perl programming*.

[OReilly06c] O'Reilly Media. *Advanced Perl programming: Accessing the Symbol Table*.

[Orfali96] R. Orfali, D. Harkey, J. Edwards. *The Essential Client / Server Survival Guide,* Second Edition". Editorial Wiley (1996).

[Orfali98] R. Orfali, D. Harkey. *Client/Server Programming with Java and CORBA,* Second Edition". Editorial Wiley (1998).

[Ortin01] F. Ortín. *Sistema Computacional De Programación Flexible Diseñado Sobre Una Máquina Abstracta Reflectiva No Restrictiva.* Tesis Doctoral. Departamento de Informática. Universidad de Oviedo (2001).

[Ortin04] F. Ortin, J. M. Cueva. *Dynamic Adaptation of Application Aspects*. Journal of Systems and Software, Volume 71, Issue 3 (2004).

[Ortin05] F. Ortin, J. M. Redondo. *Extending Rotor with Structural Reflection to support Reflective Languages*. RFP Capstone Workshop 2005 (Redmond) (2005).

[Ortin05b] F. Ortín, J. M. Redondo, J. M. Cueva. *Adding Structural Reflection to the SSCLI*. Journal of .Net Technologies, Volume 3, Number 1-3 (2005).

[Ousterhout98] J. K. Ousterhout. *Scripting: Higher Level Programming for the 21st Century*. IEEE Computer magazine (1998).

[Padrón04] J. Padrón L., A. Estévez G., J.L. Roda G., F. García L. *BOA, un framework MDA de alta productividad*. I Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones. DSDM'04, Málaga, España (2004).

[Parnas72] D. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, Vol. 15, No. 12 (1972).

[Parrot06] The Perl Foundation. *Parrot Virtual Machine - parrotcode*.

[ParrotCode06] Parrotcode.org. *Parrot JIT Subsystem*.

[Peak06] Telecommunity.com. *Peak, Python Enterprise Application Kit*.

[Pelechano04] V. Pelechano, A. Vallecillo, J. Muñoz, J. Fons. *Actas del I Taller sobre Desarrollo Dirigido por Modelos, MDA y Aplicaciones*. DSDM'04, Málaga, España (2004).

[PHP06] PHP.net. *PHP: Hypertext Processor*.

[PHPBB06] PHPBB.com. *PHPBB: Creating Communities*.

[Pinto01] M. Pinto, M. Amor, L. Fuentes, J. M. Troya. *Run-Time Coordination of Components: Design Patterns vs. Component & Aspect based Platforms*. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Advanced Separation of Concerns (2001).

[Pinto02] M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya. *Separation of Coordination in a Dynamic Aspect Oriented Framework*. AOSD 2002 Proceedings (2002).

[Pitman87] T. Pittman. *Two-level hybrid interpreter/native code execution for combined space-time program efficiency*. Proceedings of the SIGPLAN Symposium on Interpreters and Interpretive Techniques.ACMPress, New York (1987).

[Piumarta98] I. Piumarta, F. Riccardi. *Optimizing direct threaded code by selective inlining*. Proceedings of PLDI '98 (1998).

[Plezbert96] M. P. Plezbert. *Continuous Compilation For Software Development And Mobile Computing*. Ph. D. Thesis. Sever Institute of Washington University (1996).

[Plezbert96b] M. P. Plezbert. *Continuous Compilation For Software Development And Mobile Computing: Experiments*. Ph. D. Thesis. Sever Institute of Washington University.

[Plezbert97] M. P. Plezbert, R. K. Cytron. *Does 'just in time' = 'better late then never'?*. Proceedings of POPL '97 (1997).

[Pobar06] J. Pobar. *Joel Pobar's CLR weblog: Hello, world...Reflection.Emit style!*.

[Popovici01] A. Popovici, T. Gross, G. Alonso. *Dynamic Homogenous AOP with PROSE*. Technical Report, Department of Computer Science, ETH Zürich, Switzerland (2001).

[Popovici02] A. Popovici, T. Gross, G. Alonso. *Dynamic Weaving for Aspect Oriented Programming*. In 1st Intl. Conf. on Aspect- Oriented Software Development, Enschede, The Netherlands (2002).

[Popovici03] A. Popovici, G. Alonso, T. Gross. *Just-In-Time Aspects: Efficient Dynamic Weaving for Java*. In 2nd International Conference on Aspect-Oriented Software Development, Boston, USA (2003).

[Pratschner06] S. Pratschner. *The Design of the .Net Compact Framework CLR*.

[Prechelt02] L. Prechelt. *Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java*. Advances in Computers, Volume 58

[Predescu06] O. Predescu. *Javascript Introspection*.

[PSF06] Python.org. *Python Software Foundation Home Page.*

[Psyco06] Sourceforge.net. *Psyco Homepage*.

[PyPy06] Codespeak.net. *PyPy Homepage*.

[Pythius06] Sourceforge.net. *The Pythius Website*.

[Python06] Python.org. *Python General Programming FAQ*.

[PyUnit04] S. Purcell. *PyUnit - The standard unit testing framework for Python*.

[QuakeC06] Gamers.org. *Quake-C Specifications v1.0.*

[Raman98] L. G. Raman. *OSI Systems and Network Management*. IEEE Communications Magazine (1998).

[Redondo06] J. M. Redondo, F. Ortín, J. M. Cueva. *Optimización de las Primitivas de Reflexión Ofrecidas por los Lenguajes Dinámicos*. VI Jornadas sobre Programación y Lenguajes PROLE '06. Sitges (Barcelona) (Spain) (2006).

[Redondo06b] J. M. Redondo, F. Ortín, J. M. Cueva. *Diseño de Primitivas de Reflexión Estructural Eficientes Integradas en SSCLI*. Jornadas en Ingeniería del Software y Bases de Datos JISBD ' 06. Sitges (Barcelona) (Spain) (2006).

[Redondo07] J. M. Redondo, F. Ortín, J. M. Cueva. *Optimizing Reflective Primitives of Dynamic Languages*. International Journal of Software Engineering and Knowledge Engineering (2007).

[Reflex06] E. Tanter. *Reflex Kernel*.

[RemotingCORBA05] Sourceforge.net. *Remoting.CORBA Project*.

[Rémy99] D. Rémy, X. Leroy, P. Weis. *Objective Caml — a general purpose high-level programming language*. ERCIM News (1999).

[Richards71] M. Richards. *The Portability of the BCPL Compiler*. *Software* Practice and Experience (1971).

[Richards79] M. Richards, C. Whitby-Stevens. *BCPL – The Language and its Compiler*. Cambridge University Press (1979).

[Ritchie78] D. M. Ritchie, B. W. Kerninghan. *The C Programming Language*. Prentice Hall (1978).

[Rivard96] F. Rivard. *A new Smalltalk kernel allowing both explicit and implicit metaclasses programming*. Workshop in Extending the *Smalltalk* Language, OOPSLA'96. San José (EE.UU.) (1996).

[Rivières84] J. des Rivières, B. C. Smith. *The Implementation of Procedurally Reflective Languages*. Proceedings of ACM Symposium on Lisp and Functional Programming (1984).

[Roddick95] J. Roddick. *A Survey of Schema Versioning Issues for Database Systems*. Information and *Software* Technology, 37 (1995).

[Rosenblum04] M. Rosenblum. *The Reincarnation of Virtual Machines*. ACM Queue vol. 2, no. 5 (2004).

[Rossum01] G. van Rossum. *Python Reference Manual*. Fred L. Drake Jr. Editor. Relesase 2.1. (2001).

[Rotor05] Microsoft.com. *Shared Source Common Language Infraestructure 1.0 Release*.

[RotorFunded04] Microsoft Research. *Rotor Funded Projects in 2004*.

[Ruby06] Ruby: A programmers best friend. The Source for *Ruby*. *Programming Ruby - The Pragmatic Programmer's Guide*.

[RubyCentral06] RubyCentral. *The Source for Ruby*.

[RubyCentral06b] RubyCentral: The Source for *Ruby*. *Programming Ruby - The Pragmatic Programmer's Guide: Reflection, ObjectSpace, and Distributed Ruby*.

[Schofield02] J. B. G. Perez-Schofield, E. G. Roselló, T. B. Cooper, M. P. Cota. *Managing schema evolution in a container-based persistent system*. Software, Practice & Experience 32(14) (2002).

[Schofield06] J. B. G. Perez-Schofield. *Zero: Sistema de programacion simple, persistente, orientado a objetos puro y basado en prototipos*.

[Schofield06b] J. B. G. Perez-Schofield. *Tutorial de manejo de Visual Zero VM*.

[Schult02] W. Schult, A. Polze. *Aspect-oriented programming with C# and .NET*. In International Symposium on Objectoriented Real-time distributed Computing (ISORC). Crystal City, VA, USA, (2002).

[Schult02b] W. Schult, A. Polze. *Dynamic Aspect-Weaving with .NET*. Workshop zur Beherrschung nichtfunktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany (2002).

[Schult03] W. Schult, A. Polze. *Speed vs. Memory Usage - An Approach to Deal with Contrary Aspects*. The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at the International Conference on Aspect-Oriented Software Development, Boston, Massachusetts (2003).

[Scummvm06] Scummvm.org. *Script Creation Utility For Maniac Mansion Virtual Machine*.

[Segura-Devillechaise03] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, J. L. Lawall. *Web cache prefetching as an aspect: towards a dynamic-weaving based solution*. AOSD 2003 Proceedings (2003).

[Segura-Devillechaise03b] M. Ségura-Devillechaise, J.-M. Menaud. *microDyner: Un noyau efficace pour le tissage dynamique d'aspects sur processus natif en cours d'exécution*. LMO2003, Hermès, Vannes (2003).

[Sirer99] E. G. Sirer, R. Grimm, A. J. Gregory, B. N. Bershad. *Design and Implementation of a Distributed Virtual Machine for Network Computers*. 17th ACM Symposium on Operating Systems Principles (SOSP'99) (1999).

[Skarra87] A. H. Skarra, S. B. Zdonik. *Type Evolution in an Object-Oriented Database*. Research Directions in Object-Oriented Programming. The MIT Press (1987).

[Skonnard01] A. Skonnard. *XML in .NET: .NET Framework XML Classes and C# Offer Simple, Scalable Data Manipulation*. Microsoft Developer Network (2001).

[SpringAOP06] SpringFramework.org. *"Spring Framework"*.

[Squeak05] Squeak.org. "*Squeak Homepage*".

[Smith82] B. C. Smith. "*Reflection and Semantics in a Procedural Language*". MIT-LCS-TR-272. Mássachusetts Institute of Technology. Cambridge (EE.UU.) (1982).

[Smith92] R. Smith, A. Sloman, J. Gibson. *PROLOG's Two- Level Virtual Machine Support for Interactive Languages*. Research Directions in Cognitive Science, v.5 (1992).

[Smith95] R. B. Smith, D. Ungar. *Programming as an Experience: The Inspiration for Self*. Sun Microsystems Laboratories (1995).

[Software06] SoftwareProjects.org. *PHP Tutorial: PHP Types*.

[Spitz06] A. Spitz, A. Ausch, D. Ungar. *Sun Microsystems Laboratories Self: The Power of Simplicity Release 4.3*

[Srinivasan03] P. Srinivasan. *Introducción a Microsoft .NET Remoting Framework*. Microsoft Corporation (Microsoft Developer Network).

[SSCLI06] Microsoft Developer Network. *Microsoft Shared Source CLI, C# and JScript License*.

[Starforce06] Starforce.com. *Starforce: Software Protection*.

[Starforce06b] Reteam.org. *"StarForce 3 - Brief insight into a hidden world."*.

[Steel60] T. B. Steel Jr. *UNCOL: Universal Computer Oriented Language Revisited*. Datamation. (1960).

[Steele90] G. L. Steele. *Common Lisp: The Language*. *Second Edition*. Digital Press (1990).

[Stroustrup94] B. Stroustrup . *The Design and Evolution of C++: 1st Edition*. Addison-Wesley Professional; 1st edition. ISBN: 0201543303 (1994).

[Stroustrup98] B. Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley. (1998).

[Stutz03] D. Stutz, T. Neward, G. Shilling. *Shared Source SSCLI Essentials*. O'Reilly. ISSN: 0-596-00351-X (2003).

[Sullivan01] G. T. Sullivan. *Aspect-oriented programming using reflection and metaobject protocols*. Communications of the ACM. Vol 44 No. 10 (2001).

[Sun89] Sun Microsystems, Inc. *NFS: The Network File System protocol specification*. RFC 1094, Network Information Center, SRI International (1989).

[Sun95] Sun Microsystems Computer Corporation. *The Java Virtual Machine Specification. Release 1.0 Beta Draft* (1995).

[Sun96] Sun Microsystems Computer Corporation . J*avaBeans™ 1.0 API Specification* (1996).

[Sun97] Sun Microsystems Computer Corporation. *Java Reflection API Documentation (1997)*

[Sun97b] Sun Microsystems Computer Corporation. *Java Remote Method Invocation Specification (RMI)* (1997).

[Sun97c] Sun Microsystems Computer Corporation. *Java <> Native Interface Specification* (1997).

[Sun97d] Sun Microsystems Computer Corporation . *Java Core Reflection. API and Specification* (1997).

[Sun97e] Sun Microsystems Computer Corporation. *Java <> Object Serialization Specification* (1997).

[Sun98] Sun Microsystems Computer Corporation. *The Java HotSpot Virtual Machine Architecture White Paper"* (1998).

[Sun99] Sun Microsystems Computer Corporation. *Dynamic Proxy Classes* (1999).

[SunComp06] Sun Developer Network. *Desktop Java: Javabeans*.

[Sundararajan06] A. Sundararajan. *Java, Groovy and (J)Ruby*. A. Sundararajan Weblog.

[SunHotSpot06] Sun Developer Network. *The Java HotSpot Virtual Machine v1.4.1*.

[SunJ2EE07] Sun Developer Network. *Java EE at a Glance*.

[SunJavadoc06] Sun Developer Network. *Core Java: Javadoc Tool*.

[SunJScript06] Sun Microsystems. *JavaScript Guide: Using Navigator Objects*.

[SunKlein06] Sun Microsystems. *Klein: New architectural forms to simplify construction of high performance or smallfootprint Java virtual machines.*

[SunPicoJ06] Sun MicroElectronics. *"picoJava Microprocessor Cores"*.

[Suraski06] Z. Suraski. *The Object-Oriented Evolution of PHP*

[Swaine94] M. Swaine *"Programming Paradigms. Developing for Newton"*. Dr. Dobb's Journal, No. 212, 115-118 (1994).

[Taivalsaari92] A. Taivalsaari. *Kevo, a prototype-based object-oriented programming lnaguage based on concatenation and module operations*. Technical Report Report LACIR 92-02, University of Victoria (1992).

[Tan89] L. Tan, T. Katayama. *Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution*. Proceedings of First International Conference on Deductive and Object-Oriented Databases, DOOD, (1989).

[Tarr99] P. Tarr, H. Ossher, W. Harrison, S. Sutton. *N Degrees of separation: Multi-Dimensional Separation of Concerns*. Proceedings of the 1999 International Conference on Software Engineering (1999).

[Tatsubori98] Michiaki Tatsubori, Shigeru Chiba. *Programming Support of Design Patterns with Compile-time Reflection*. OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canada) (1998).

[Tcl06] Tcl.tk. *Tcl Developer Xchange: Tcl features*.

[Thibault00] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, G. Muller. *Static and dynamic program compilation by interpreter specialization*. Higher-Order Symbol. Computat (2000).

[Thomas05] D. Thomas, D. Heinemeier Hansson. *Agile Web Development with Rails*. A Pragmatic Guide. Pragmatic Bookshelf (2005).

[Tldp06] tldp.org. *Guia del Usuario de Ruby: Clases*.

[Trados96] A. Trados, E. Uber. *Using Borland's Delphi and C++ Together*. A technical Paper for Developers. Borland Online (1996).

[Turing36] A. M. Turing. *On Computable Numbers, with an Application to the Entscheidungs problem*". Proceedings of The London Mathematical Society, Series 2, 42 (1936).

[Ung00] D. Ung, C. Cifuentes. *Machine adaptable dynamic binary translation*. Proceedings of Dynamo '00 (2000).

[Ungar87] D. Ungary R. B. Smith. "*SELF: The Power of Simplicity*". In OOPSLA'87 Conference Proceedings. Published as SIGPLAN Notices, 22, 12, 227-241. 1987.

[Ungar91] D. Ungar, C. Chambers, B. Chang, U. Hölzle. *Organizing Programs Without Classes*. Lisp and Symbolic Computation: An International Journal, 4, 3 (1991).

[VanRoy94] P. Van Roy. *The wonder years of sequential Prolog implementation*. J. Logic Program (1994).

[Vasseur04] A. Vasseur. *Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address it?*. Proceedings of the Dynamic AOP Workshop in conjunction with 3rd International Conference on Aspect-Oriented Software Development AOSD (2004).

[Venners98] B. Venners. *Inside the Java Virtual Machine*. *Java* Masters. McGraw Hill (1998).

[Venners03] B. Venners. *Orthogonality and the DRY Principle.* Artima Developer (2003).

[Vinuesa03] L. Vinuesa M. *Separación dinámica de aspectos independiente del lenguaje y plataforma, mediante reflectividad computacional*. Trabajo de Investigación para el programa de doctorado "Avances en Informática". Universidad de Oviedo (2003).

[Vinuesa07] L. Vinuesa M. *Separación dinámica de aspectos independiente del lenguaje y plataforma mediante el uso de reflexión computacional*. Tesis Doctoral. Universidad de Oviedo (2007).

[VisualStudio06] Microsoft.com. *Microsoft Visual Studio .NET*.

[Vo96] K.-P. Vo. *Vmalloc: A General and Efficient Memory Allocator*. *Software* – Practice and Experience (1996).

[Vogels05] W. Vogels. *All things Distributed*.

[Volkmann03] M. Volkmann. *Groovy - Scripting for Java*. Object Computing, Inc. (OCI) (2003).

[W3C98] WWW Consortium. *Extensible Markup Language (XML) 1.0* (1998).

[W3CXSLT99] WWW Consortium. *XSL Transformations (XSLT) Version 1.0*.

[W3DOM06] WWW Consortium. *Document Object Model (DOM)*.

[W3Schools06] W3Schools.com. *JavaScript: Create Your Own Objects*.

[Wall00] L. Wall, T. Christiansen, J. Orwant. *Programming Perl 3rd Edition*. O'Reilly. ISBN 0-596-00027-8 (2000).

[Wall00b] L. Wall, T. Christiansen, J. Orwant. *Programming Perl 3rd Edition: Chapter 12. Objects*. O'Reilly. ISBN 0-596-00027-8 (2000).

[Wand88] M. Wand, D. P. Friedman. *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower*. Meta-Level Architectures and Reflection. P. Maes, D. Nardi Editors. North-Holland (1988).

[Watanabe88] T. Watanabe, A. Yonezawa. *Reflection in an object-oriented concurrent language*. Proceedings of OOPSLA'88, vol 23. SIGPLAN Notices, ACM Press (1988).

[WebReference06] Webreference.com. *Object-Oriented Programming with JavaScript, Part I: Inheritance. Inheritance through Functions.*

[Welch98] I. Welch, R. Stroud. *Dalang – A Reflective Java Extension*. OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá) (1998).

[Whitfield97] D. L. Whitfield, M. L. Soffa. *An approach for exploring code improving transformations*. ACM TOPLAS (1997).

[WikiDNET06] Wikipedia, the free encyclopedia. *.NET Framework: Diagrama de la plataforma .NET*.

[WikiDNET06b] Wikipedia, the free encyclopedia. *.NET Framework: Diagrama de funcionamiento de CLI*.

[WikiDNET06c] Wikipedia, the free encyclopedia. *.NET Framework: Diagrama de la arquitectura de .NET Framework 3.0*.

[WikiMono06] Wikipedia, the free encyclopedia. *Diagrama simplificado de la plataforma Mono*.

[WinScript06] Microsoft Developer Network. *Windows Script*.

[Wolczko96] M. Wolczko, O. Agesen, D. Ungar. *Towards a Universal Implementation Substrate for Object-Oriented Languages*. Sun Microsystems Laboratories, documento #96-0506 (1996).

[Wolczko06] M. Wolczko, R. B. Smith. *Prototype-Based Application Construction Using SELF 4.0.*

[Wrenholt05] E. Wrenholt. *Ruby, Io, PHP, Python, Lua, Java, Haskell and Plain C Fractal Benchmark* (2005).

[Xdoc2005] SourceForge.net. *Xdoclet attribute oriented programming*.

[Xerox05] Parc.com. *Xerox Palo Alto Research Center*.

[XOTcl06] XOTcl - Extended Object Tcl Homepage. *XOTcl Tutorial*.

[Yoder01] J. W. Yoder, F. Balaguer, R. Johnson. *Architecture and Design of Adaptive Object Models.* Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA, ACM SIGPLAN Notices (2001).

[Yokote92] Y. Yokote. *The New Mechanism for Object-Oriented System Programming*. Proceedings of IMSA'92 International Workshop on Reflection and Meta-level Architecture (1992).

[Yonezawa90] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Computer Science Series. The MIT Press (1990).

[Zdun06] U. Zdun, G. Neumann. *XOTcl - Extended Object Tcl*.

[Zhao2003] M. Zhao, B. Childers, M.L. Soffa. *Predicting the impact of optimizations for embedded systems*. Proceedings of LCTES 2003 (2003).

[Zheng2000] C. Zheng, C. Thompson. *PA-RISC to IA-64: Transparent execution, no recompilation*. IEEE Computer (2000).

[Zimmer98] J. A. Zimmer. *Tcl/Tk for Programmers*. IEEE Computer Society. John Wiley and Sons, ISBN 0-8186-8515-8 (1998).

[Zinki97] J.A. Zinky, D. E. Bakken, R. E. Schantz. *Architectural Support for Quality of Service for CORBA Objects*. Theory and Practice of Object Systems, 3 (1997).

[ZipLib05] ICSharpcode. *Zip Lib Homepage*.

[Zope06] Zope Community. *Zope.org: The Web Site for the Zope Community*.

[Zorn06] B. Zorn. *Ben Zorn CLI Benchmarks*. Microsoft Research.