JSS 7422
9 December 2002 Disk used
ARTICLE IN PRESS
No. of Pages 15, DTD = 4.3.1
SPS-N, Chennai

ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

The Journal of Systems and Software xxx (2002) xxx–xxx

The Journal of Systems and Software

www.elsevier.com/locate/jss

Dynamic adaptation of application aspects

Francisco Ortin *, Juan Manuel Cueva [1]

Department of Computer Science, University of Oviedo. Calvo Sotelo s/n, Oviedo 33007, Spain

Received 24 April 2002; received in revised form 25 November 2002; accepted 25 November 2002

Abstract

In today's fast changing environments, adaptability has become an important feature in modern computing systems, programming languages and software engineering methods. Different approaches and techniques are used to achieve the development of adaptable systems. Following the principle of separation of concerns, aspect-oriented programming (AOP) distinguishes application functional code from specific concerns that cut across the system, creating the final application by *weaving* the program's main code and its specific aspects. In many cases, dynamic application adaptation is needed, but few existing AOP tools offer it in a limited way. Moreover, these tools use a fixed programming language: aspects cannot be implemented regardless of its programming language.

We identify reflection as a mechanism capable of overcoming the deficiencies previously mentioned. We have developed a non-restrictive reflective technique that achieves a real computational-environment jump, making every application and language feature adaptable at runtime—without any previously defined restriction. Moreover, our reflective platform is independent of the language selected by the programmer. Using the reflective capabilities of the platform presented, an AOP framework that achieves dynamic aspect weaving in a language-independent way has been constructed, overcoming the common limitations of existing AOP tools.

Keywords: Aspect-oriented programming; Reflection; Separation of concerns; Dynamic weaving; Meta-object protocol

1. Introduction

In many cases, significant concerns in software applications are not easily expressed in a modular way. Examples of such concerns are transactions, security, logging or persistence. The code that addresses these concerns is often spread out over many parts of the application. Software engineers have used the principle of separation of concerns (Parnas, 1972; Hürsch and Lopes, 1995) to manage the complexity of software development; it separates main application algorithms from special purpose concerns. Final applications are built by means of its main functional code plus their specific problem-domain concerns. The main benefits of this principle are:

1. Higher level of abstraction, since the programmer can reason about individual concerns in isolation.
2. Easier to understand the application functionality. The application's source code is not cluttered with the code of other concerns.
3. Concern reuse. Separation of concerns attains decoupling of different modules, achieving reusability of single concerns.
4. Increase of application development productivity. In addition to previously mentioned advantages, the use of testing and debugging concerns (such as tracing, pre and post condition contract enforcement or profiling) might facilitate the application construction—without needing to modify the functional source code.

This principle has been performed following several approaches. Aspect-oriented programming (AOP) (Kiczales et al., 1997), multi-dimensional separation of concerns (Tarr et al., 1999) or reflective meta-object protocol (MOP) programming languages (Kiczales et al., 1992), are well-known examples.

* Corresponding author. Tel.: +34-985-10-3172; fax: +34-985-10-3354.
E-mail addresses: ortin@lsi.uniovi.es (F. Ortin), cueva@pinon.ccu.uniovi.es (J. Manuel Cueva).
[1] Tel.: +34-985-10-3286, fax: +34-985-10-3354.

0164-1212/02/$ - see front matter © 2002 Published by Elsevier Science Inc.
PII: S0164-1212(02)00157-7

Both AOP and multi-dimensional separation of concerns achieve the construction of concern-adaptable programs. Most existing tools lack adaptability at runtime: once the final application has been generated (*woven*), it will not be able to adapt its concerns (aspects) at runtime. There are certain cases in which the adaptation of application concerns should be done dynamically, in response to changes in the runtime environment—e.g., distribution concerns based on load balancing (Matthijs et al., 1997).

To overcome the static-weaving tools limitations, different dynamic-weaving AOP approaches—like AOP/ST (Böllert, 1999), PROSE (Popovici et al., 2001) or Dynamic Aspect-Oriented Platform (Pinto et al., 2001)—have appeared. However, as we will explain in Section 2, they limit the set of join points they offer, restricting the way aspects can be adapted at runtime. Another drawback of existing tools is that they use fixed programming languages: aspects and concerns are not reusable regardless of its programming language.

Reflection is a programming language technique that achieves dynamic application adaptability. It can be used to reach aspect adaptation at runtime. Most runtime reflective systems are based on the ability to modify the programming language semantics while the application is running (e.g., the message passing mechanism). However, this adaptability is commonly achieved by implementing a protocol (MOP) as part of the language interpreter that specifies—and therefore, restricts—the way a program may be modified at runtime. As we will explain in Section 3.1, other limitations of common MOP-based systems are their language-dependence and their restrictions expressing system's features modification.

We have developed a non-restrictive reflective platform called nitrO (Ortin and Cueva, 2002), in which it is possible to change any programming language and application feature at runtime, without any kind of restriction imposed by an interpreter protocol. Our platform achieves language neutrality: any programming language can be used, and every application is capable of adapting another one's characteristic, no matter whether they use the same programming language or not.

By using nitrO as the back-end of our AOP system, it is possible to develop dynamic modification of application aspects. Applications may dynamically adapt their concerns to unpredictable design-time requirements, changing them at runtime—without any previously defined restriction.

The rest of this paper is structured as follows. In Section 2 we present AOP and the main lacks of existing tools. Section 3 briefly describes two reflection classifications as well as MOP advantages and drawbacks; we also present the reflective features of the Python programming language. Section 4 introduces our system architecture; its design is presented in Section 5. How application and programming languages are represented is described in Section 6, and different dynamic aspect-adaptation examples are shown in the following section. Finally, we analyze runtime performance (Section 8) and Section 9 presents the ending conclusions.

2. Aspect-oriented programming

AOP technique (Kiczales et al., 1997) provides explicit language support for modularizing application concerns that crosscut the application functional code. Aspects express functionality that cuts across the system in a modular way, thereby allowing the developer to design a system out of orthogonal concerns and providing a single focus point for modifications. By separating the application functional code from its crosscutting aspects, the application source code would not be tangled, being easy to debug, maintain and modify (Parnas, 1972).

Application persistence, tracing or synchronization policy, are examples of aspects that can be used in different applications, whatever its functionality would be. Aspect-oriented tools create programs combining the application functional code and its specific aspects. The process of integrating the aspects into the main application code is called *weaving* and a tool called *aspect weaver* performs it.

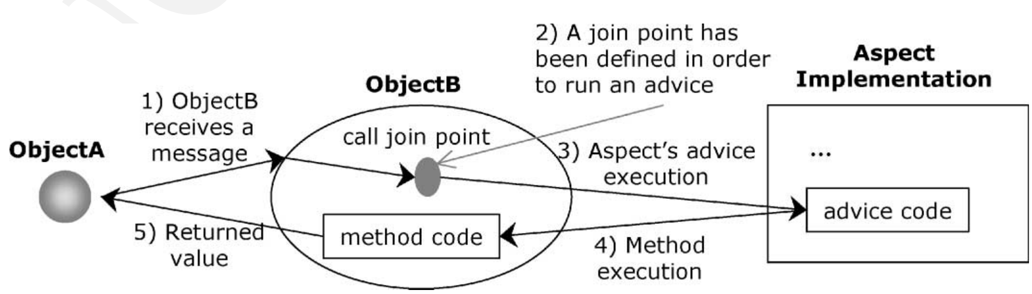

Fig. 1. Separating functional code from specific aspects.

2.1. Static weaving

Most current AOP implementations are largely based on static weaving: compile-time modification of application source code, inserting calls to specific aspect routines. The places where these calls are inserted are called *join points*.

AspectJ (Kiczales et al., 2001) is an example of a static-weaving aspect-oriented tool: a general-purpose aspect-oriented extension to Java that supports AOP. The way AspectJ supports aspect-oriented separation of concerns is by following the next steps (as shown in Fig. 1):

1. Identifying join points in the application's functional code by means of *pointcuts designators*. We must identify certain well-defined points in the execution of a program where calls to aspect code would be inserted. An example of a common join point is a method call.
2. Implementing *advice* to be run at join points. This code will be executed when a join point is reached, either before or after the computation proceeds.
3. Declaring *aspects*. An aspect is a modular unit of crosscutting implementation that is provided in terms of pointcuts and advice, specifying *what* (advice) and *when* (pointcut) its code is going to be executed.
4. Generating the final application. The AspectJ compiler "`ajc`" (O'Brien, 2001) takes both the application functional code and its specific aspects, producing the final Java2 ".class" files.

2.2. Dynamic weaving

Using a static weaver, the final program is generated by weaving the application functional code and its selected aspects. If we want to enhance the application with a new aspect, the system has to be re-compiled and re-started.

Although not every application aspect needs to be adapted at runtime, there are specific aspects that will benefit from a dynamic-weaving system. There could be applications that need to dynamically adapt its specific concerns in response to changes in the runtime environment (Popovici et al., 2001). As an example, related techniques has been used in handling Quality of Service (QoS) requirements in CORBA distributed systems (Zinky et al., 1997).

In order to overcome the static-weaving weaknesses, different dynamic-weaving approaches have emerged: e.g. AOP/ST (Böllert, 1999), PROSE (Popovici et al., 2001) or Dynamic Aspect-Oriented Platform (Pinto et al., 2001). These systems offer the programmer the ability to dynamically modify the aspect code assigned to application join-points—similar to runtime reflective systems (Maes, 1987).

The limited set of language join-points restricts the amount of application features an aspect can adapt. For instance, PROSE cannot implement a post-condition-like aspect, since its join-point interface does not allow accessing the value returned by a method upon exit (Popovici et al., 2001).

We think that an interesting dynamic-weaving issue is giving a system the ability to adapt to runtime-emerging aspects unpredicted at design time—e.g., a logging aspect not considered previously to the application execution. A system that offers a limited set of join points restricts this facility.

2.3. Language dependency

Both static and dynamic weaving AOP tools do not offer the implementation of crosscutting concerns, regardless of the language that the programmer might use. They use fixed-language techniques to achieve separation of concerns.

We have identified computational reflection (Maes, 1987) as the best technique to overcome the previously mentioned limitations. In this paper, we present our reflective and language-neutral programming platform employed to achieve dynamic and non-restrictive aspect adaptation, in a language-independent way.

3. Categorizing reflection

We identify two main criteria to categorize reflective systems. These criteria are *when* reflection takes place and *what* can be reflected. If we take *what* can be reflected as a criterion, we can distinguish:

- *Introspection*: The system's structure can be accessed but not modified. If we take Java as an example, with its "`java.lang.reflect`" package, we can get information about classes, objects, methods and fields at runtime.
- *Structural reflection*: The system's structure can be modified. An example of this kind of reflection is the addition of object's fields—attributes.
- *Computational (behavioral) reflection*: The system semantics (behavior) can be modified. For instance, metaXa—formerly called MetaJava (Kleinöder and Golm, 1996)—offers the programmer the ability to dynamically modify the method dispatching mechanism.

Taking *when* reflection takes place as the classification criterion, we have:

- *Compile-time reflection*: The system customization takes place at compile-time—e.g., OpenJava (Chiba and Michiaki, 1998). The two main benefits of this kind of systems are runtime performance and the ability to adapt its own language. Many static-weaving aspect-oriented tools use this technique.

- *Runtime reflection*: The system may be adapted at runtime, once it has been created and run—e.g., metaXa. These systems have greater adaptability by paying performance penalties.

Our system, nitrO (Ortin and Cueva, 2001), achieves *computational* reflection at *runtime*. Moreover, our reflection technique implementation is more flexible than common runtime reflective systems—as we will explain in the next section—and it is not language-dependent.

3.1. Meta-object protocols restrictions

Most runtime reflective systems are based on MOPs (MOPs). A MOP specifies the implementation of a reflective object-model (Kiczales et al., 1992). An application is developed by means of a programming language (base level). The application's meta-level is the implementation of the computational object model supported by the programming language at the interpreter computational environment. Therefore, a MOP specifies the way a base-level application may access its meta-level in order to adapt its behavior at runtime.

As shown in Fig. 2, the implementation of different meta-objects can be used to override the system semantics. For example, in MetaXa (Kleinöder and Golm, 1996), we can implement the class "Trace" inherited from the class "MetaObject" (offered by the language as part of the MOP) and override the "eventMethodEnter" method. Its instances are meta-objects that can be attached to user objects by means of its inherited "attachObject" message. Every time a message is passed to these user objects, the "eventMethodEnter" method of its linked meta-objects will be called—showing a trace message and, therefore, customizing its message-passing semantics.

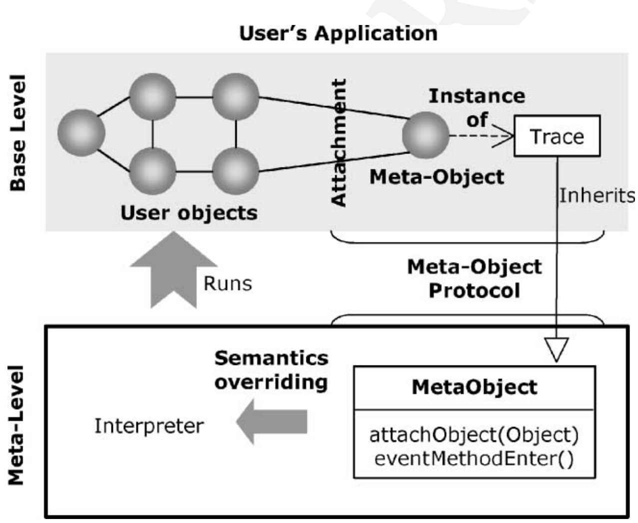

Fig. 2. MOP-based program adaptation.

The MOP reflective technique has different drawbacks:

1. The way a MOP is defined restricts the amount of features that can be customized (Douence and Südholt, 1999). If we do not consider a system feature to be adaptable by the MOP, this application's attribute will not be able to be customized once the application is running. In our example, if we would like to adapt the way objects are created, we must stop the program execution and modify the MOP implementation.
2. Changing the MOP would involve different interpreter and language versions and, therefore, previous existing code could result deprecated.
3. The way a semantic feature may be customized has expressiveness restrictions. Objects behavior may be overridden by attaching meta-objects to them. These meta-objects express how they modify the objects' behavior by just overriding its super-class' methods—it follows the Template Method design pattern (Gamma et al., 1995). The use of a whole meta-language would be a richer mechanism to express the way an application may be adapted.
4. Finally, MOP-based systems are language-dependent. They do not offer runtime adaptability in a language-independent way.

Some advanced dynamic-weaving AOP tools, like PROSE (Popovici et al., 2001), use MOP-based reflective interpreters on its back-ends. Therefore, this kind of dynamic separation of crosscutting concerns will not be capable of overcoming these four disadvantages.

Our nitrO runtime reflection mechanism is based on a meta-language specification (Ortin and Cueva, 2002). The way the base level accesses the meta-level (reification) is specified by another language (meta-language)—not by using a MOP. The meta-language is capable of adapting the structure and behavior of the base level at runtime, without any restriction and independently of the language being used. Its design will be specified in Section 4.

3.2. Python's reflective capabilities

We have selected the Python programming language (Rossum, 2001) to develop our system because of its reflective capabilities (Andersen, 1998):

- *Introspection*: At runtime, any object's attribute, class or inheritance graph can be inspected. It can also be inspected the dynamic symbol table of any application: its existing modules, classes, objects and variables at runtime.

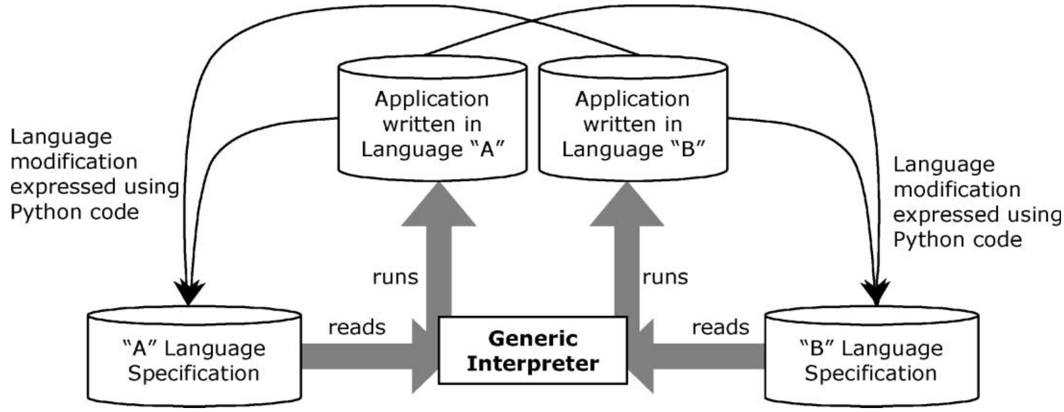

Fig. 3. System architecture.

- *Structural reflection*: It is possible to modify the set of methods a class offers and the set of fields an object has. We can also modify the class an object is instance of, and the set of super-classes a class inherits from.
- *Dynamic evaluation of code represented as strings*. Python offers the "exec" function that evaluates a string as a set of statements. This feature can be used to evaluate code generated at runtime.

4. System architecture

The theoretical definition of reflection uses the notion of a reflective tower (Smith, 1984): we have a tower in which an interpreter, that defines its operational semantics, is running the user program. A reflective computation is a computation about the computation, i.e. a computation that accesses the interpreter. If an application would be able to access its interpreter at runtime, it would be capable of inspecting runtime system objects (introspection), modifying its structure (structural reflection) and customizing its language semantics (computational reflection).

Our reflective platform follows this scheme, allowing applications to access the interpreter computational environment. Opposite to MOP-based systems, a real computational-environment jump gives the programmer the ability to dynamically get into and modify any application and language feature. However, this mechanism is difficult to implement. Interpreters commonly have complex structures representing different functionality like parsing mechanism, semantics interpretation, or runtime user-application representation. For instance, modifying by error the parsing mechanism would involve unexpected results.

What we have developed is a generic interpreter that separates the structures accessible by the base level from the fixed modules that should never be modified. This generic interpreter is language-neutral: its inputs are both the user application and the language specification. It is capable of interpreting any programming language by reading its specification, as shown in Fig. 3.

At runtime, any application may access its language specification (or another one's language) by using the whole expressiveness of the Python programming language. There are no pre-established limitations imposed by either an interpreter protocol or a set of join-points: any language feature can be adapted. Changes performed in a programming language are automatically reflected on the application execution, because the generic interpreter relies on the language specification while the application is running.

5. System design

In Fig. 4, we show how the generic interpreter, every time an application is running, offers two sets of objects to the reflective system: the first one is the language specification represented as a graph of objects (we will explain its structure in the next section); the second group of objects is the runtime application's symbol table: variables, objects and classes created by the user.

Any application may access and modify these object structures by using the Python programming language; its reflective features will be used to:

1. If an application symbol table is inspected, introspection between different applications (independently of the language used) is achieved.
2. Modifying the symbol table structure, by means of Python structural reflective capabilities, implies structural reflection of any running application.
3. If the semantics of a language specification is modified, customization of its running application's behavior is achieved (computational reflection).

5.1. Computational jump

The main question of this design is how the application computational environment may access and

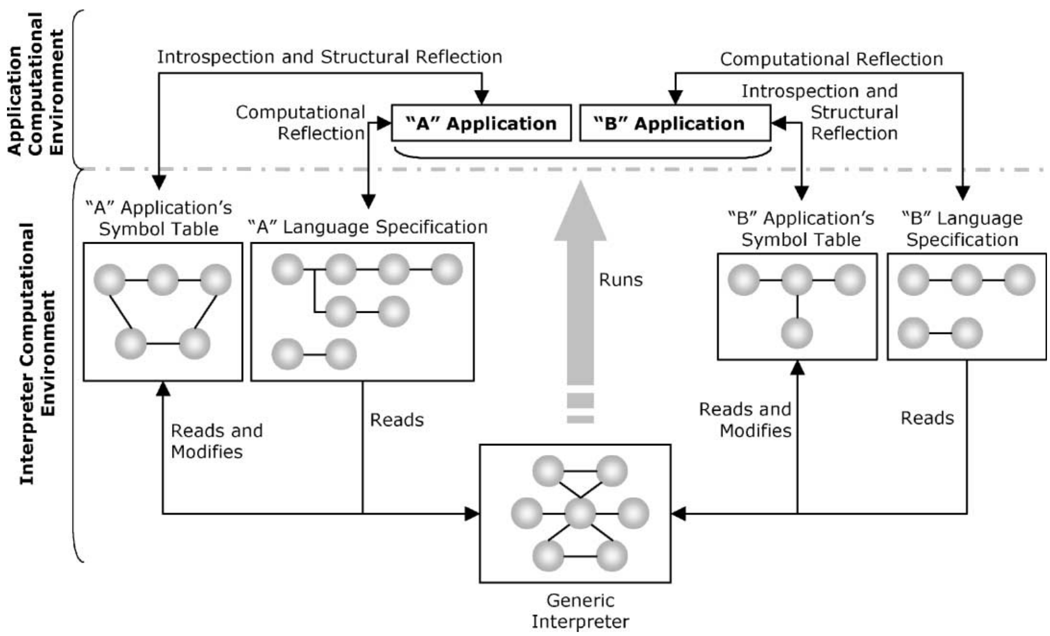

Fig. 4. Dynamic language specification and symbol-table access.

modify the interpreter computational environment—i.e., different language specifications and application symbol tables.

Every language in our system includes the "reify" statement; the generic interpreter automatically recognizes it, no matter the language being used. Inside the reify statement, Python code can written. This Python code will not be processed as the rest of the application code: every time the interpreter recognizes a reify statement, its Python code will be taken and evaluated by invoking the "exec" function. This Python code, using Python structural reflection, may access and modify application symbol tables and language specifications. This scheme is shown in Fig. 5.

The code written inside a "reify" statement is evaluated in the interpreter computational environment, not in the application computing-environment—the place where it was written. So, Python becomes a meta-lan-

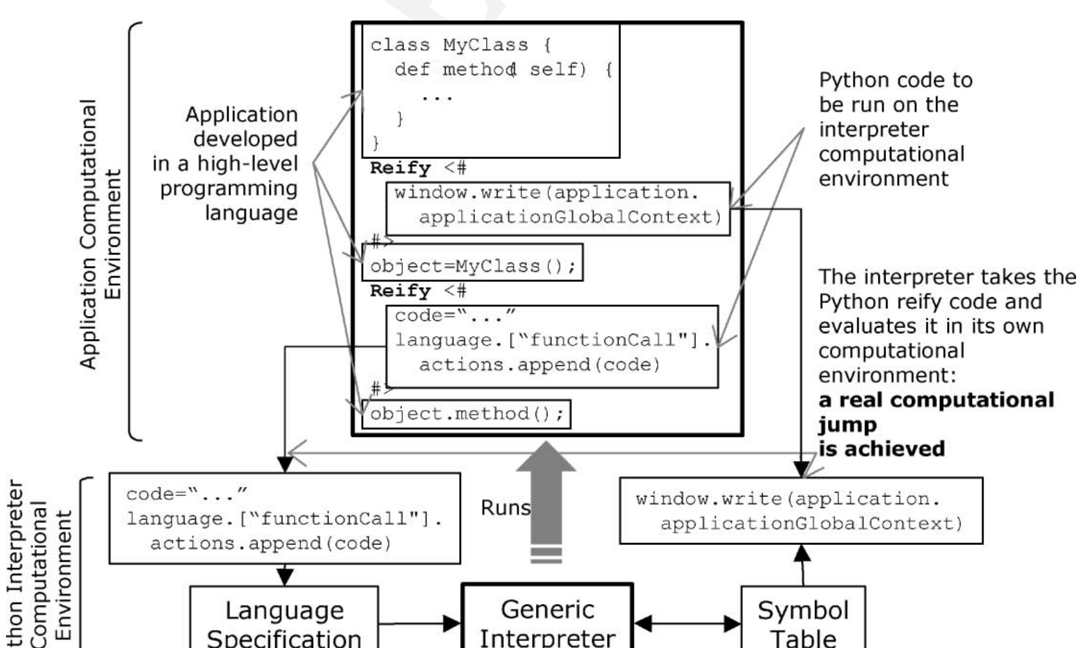

Fig. 5. Computational environment jump.

guage to specify, and dynamically modify, any language and application that would be running in the system. There is no need to specify either application join-points or a protocol that would previously restrict *what* system features could be adapted.

Python code inside a "reify" statement might be written improperly, having syntax or semantic errors. The correctness verification of these Python statements is done by the "exec" function raising an exception. Consequently, the programmer may handle this exception knowing whether the reify Python code has been executed correctly or not.

6. Language and application description

As we have seen in the previous section, applications in our system may dynamically access language specifications and application symbol tables, in order to achieve different levels of reflection. What we present in this point is how languages and applications are represented by means of object structures.

Programming languages are specified with meta-language files. Their lexical and syntactic features are expressed by means of context-free grammar rules; their semantics, expressed in Python code, are placed at the end of each rule. We have already specified the Python programming language and some domain-specific languages; currently, we are writing the Java and Jscript specifications. Correctness verification (e.g., type checking) is expressed using Python code as part of the semantic actions; these semantic-analysis routines make extensive use of application symbol and type tables.

The next code shows part of the "MetaPython" programming language—a meta-language specification of a subset of the Python programming language:

```
Language = MetaPython
Scanner = {
  "Digit Token"
    digit → "0" | "1" | "2" | "3" | "4" | "5" | "6" |
    "7" | "8" | "9";
  "Number Token"
    NUMBER → digit moreDigits;
  "Zero or more digits token"
    moreDigits → digit moreDigits
     |;
  ...
}
Parser = {
  "Initial Context-Free Rule"
    S → statement moreStatements SEMI-
    COLON <#
# Application execution initialization
global classes, functions, classAnalysed,
functionAnalised, functionResult
classes = { } # Classes Symbol Table
functions = { } # Function Symbol Table
...
#> ;
  "Zero or more Statements"
    moreStatements → statement more-
    Statements <#
nodes[2].execute()
nodes[3].execute()
#>
      |;
  "Statement"
    statement → classDefinition <#
nodes[1].execute()# Inserts the class
into the ST
#>
  ...
    | _REIFY_ <#
  nodes[1].execute()
#>
     ...
     ;
  ...
  "Method or function call"
    functionCall → ID OPENBRACE args
    CLOSEBRACE <#
...
#>
        | ID DOT ID OPENBRACE args
  CLOSEBRACE <#
...
#>
    ;
  ...
}
```

Lexical rules are specified in the "Scanner" section. Syntactic ones are located in the "Parser" scope. At the end of each rule, Python code can be placed representing language semantics. Ellipsis points in the sample meta-language grammar indicate elements deliberately suppressed—the whole language specification can be obtained from `http://www.di.uniovi.es/reflection/lab/prototypes.html#nrrs`.

The "_REIFY_" reserved word indicates where a reify statement may be syntactically located. Every application file must indicate its programming language previously to its source code. When the application is about to be executed, its respective language specification file is analyzed and translated into an object representation.

"Non-Terminal" objects, symbolizing non-terminal symbols of the rule's left-hand side, represent each language production. These objects are associated to a group of "Right" objects, which represent the rule's right-hand sides. A "Right" object has two attributes:

1. *Attribute "nodes"*: Collects "Terminal" and "NonTerminal" objects representing the production's right-hand side.
2. *Attribute "actions"*: List of "SemanticAction" objects; each of them stores the Python code located at the end of each rule specification. This code will be executed in the application interpretation.

Fig. 6 shows a fragment of the object diagram representing the example shown above.

Any application code starts with its unique ID ("Bank App" in the next example) followed by its language name ("MetaPython"). The language can also be specified inside the application file, using the meta-language. In that case, the system will be capable of running the application even though it does not hold its language specification. This is a MetaPython sample application:

```
Application = "Bank App"
Language = "MetaPython"
import string;
import random;
class Account {
  def init(self, user, credit){
    self.user = user;
    self.credit = credit;
    }
  def withdraw(self, ammount){
    self.credit = self.credit-ammount;
    return ammount;
    }
  def creditTransfer(self, ammount){
    self.credit = self.credit+ammount;
    }
}
account = Account( );
account.init('myself', 2000);
while 1 {
  if random.random( )<0.5{
    account.creditTransfer(100);
    print('Transfer done!');
  }
  else{
    account.withdraw(100);
    print('Withdraw done!');
  }
}
```

The code above simulates a simple bank application. It first defines a class, creates an instance, and sends it two messages at random in an infinite loop. The object has two fields that store the identity of the account owner and her credit.

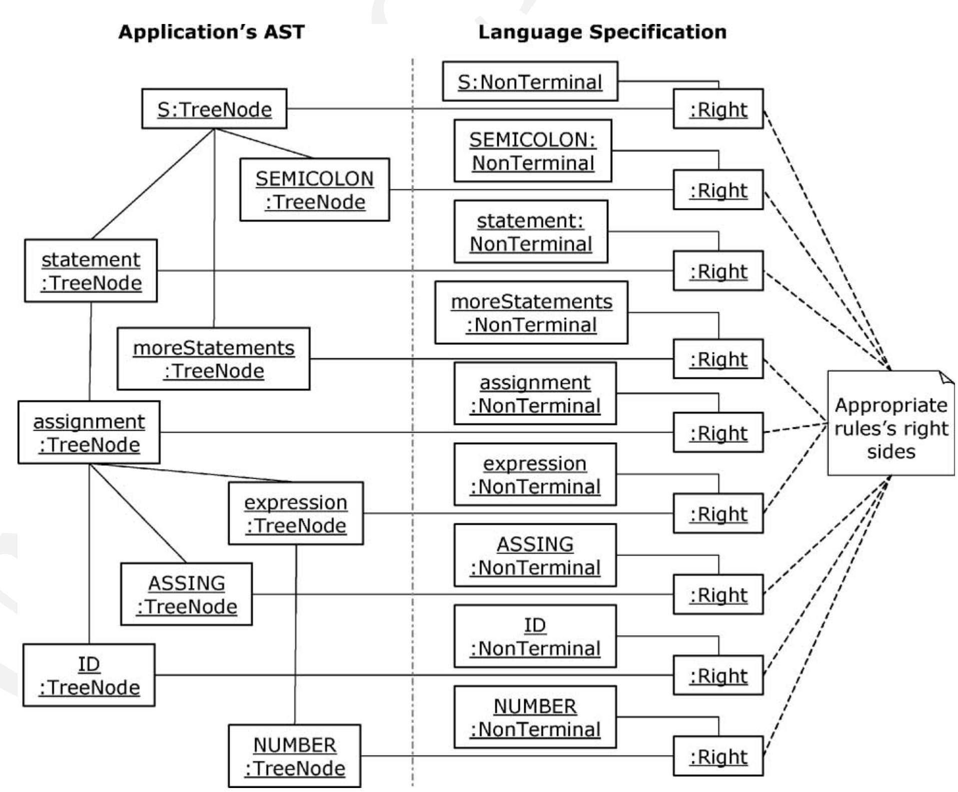

Fig. 6. Fragment of the language specification object diagram.

Once the application's language specification has been translated into its respective object structure, a backtracking algorithm parsers the application's source code creating its abstract syntax tree (AST). Then, the initial non-terminal's code is executed. The tree-walking process is defined by the way grammar-symbol's "execute" methods are invoked: the non-terminal "execute" method evaluates its associated semantic action. This way, the AST nodes are connected with its language specification structure (Fig. 6); changes on the language specification will automatically be reflected on the application execution.

Interoperability between different applications programmed in different languages is achieved by accessing the "nitrO" global object. Its attribute "applications" is a hash table that collects every existing application in the system. Each application object has two attributes:

1. *Attribute "language"*: Its language specification.
2. *Attribute "applicationGlobalContext"*: Its dynamic symbol table.

7. Dynamic aspect adaptation

Accessing the "nitrO" object attributes, any application can adapt another one's behavior and structure at runtime, without any predefined restriction and in a language-independent way. Dynamic language semantics customization can be used to change application aspects at runtime, not needing to specify its join-points at the time they are being implemented.

Introspective and structural reflective features of our platform give the programmer the opportunity to easily access and modify runtime objects in order to develop reusable and generic aspects such as persistence or distribution (Foote, 1992). As a first example, we can use introspection to develop a trace routine that shows any application runtime symbol table, regardless of its programming language:

```
[1] Application = "Trace Symbol-Table Aspect"
[2] Language = <#
[3] Language = JustReflection
[4] Scanner = { }
[5] Parser = {
[6]   "Initial Free-Context Rule"
[7]     S → _REIFY_<#
[8] nodes[1].execute( )
[9] #>; }
[10] Skip = {"\n";"\t";""; }
[11] NotSkip = { }
[12] #>
[13] reify<#
[14] # weave is the aspect-weaving routine
[15] def weave(self, appID):
[16]   # Is the appID application running?
[17]   if self.nitrO.apps.has_key(appID):
[18]     theApp = self.nitrO.apps[appID]
[19]     # Shows the Symbol Table in the aspect window
[20]     self.window.write(theApp.applica-
     self.window.write(theApp.applica-
     w.write(theApp.applicationGlobalCon-
     plicationGlobalContext)
[21]   else:
[22]     self.nitrO.shell.write("The application named \""+appID+"\"must be started.\n")
[23] nitrO.apps["Trace Symbol-Table Aspect"]._class_.weave = weave
[24] write("Routine installed as the \"weave\"method of \"TraceSymbol-Table Aspect\"application.\n")
[25] #>
```

This application specifies itself its own programming language: "JustReflection" (lines 2–12), a unique "reify" statement (lines 13–25). If we run this application, a dynamic aspect that shows any program's symbol table is installed in the system—the message "*Routine installed as the "weave" method of "Trace Symbol-Table Aspect" application*" is shown (line 24). The reify statement defines a function (line 15) and afterwards sets it as an application method (line 23). This method takes a program ID as a parameter and searches its application object in the system (lines 17 and 18). If it is found, the application symbol table will be displayed in the aspect graphic window (line 20).

Any running program's symbol-table could be shown using this aspect, regardless of the language it has been written in. For instance, we can show the "Bank App" application symbol table executing the next statement in the nitrO shell:

```
nitrO.apps["Trace Symbol-Table Aspect"].
weave("Bank App")
```

The previous aspect shows the whole runtime symbol table of any application, written in any language. If we just want to trace the existing user classes of any application, we should take into account its programming language. The localization of user classes within a symbol table depends on the way the language has been specified.

In order to suppress any language-specific dependency in every aspect implementation, a set of facilities have been implemented in the "aspectFacilities" module. These routines make the development of crosscutting concerns easier, offering the aspect programmer language-independent facilities. As an example, the following aspect code employs the language-neutral "getClassesFromSymbolTable" function, which returns the list of existing classes in an application's symbol-table, whatever its language might be.

```
Application="Show Classes Aspect"
Language=<#
Language=JustReflection
Scanner={}
Parser={
  "Initial Free-Context Rule"
    S→_REIFY_<#
nodes[1].execute()
#>;}
Skip={"\n";"\t";"";}
NotSkip={}
#>
reify<#
def weave(self,appID):
  if self.nitrO.apps.has_key(appID):
    from aspectFacilities import*
    import aspectFacilities
    app=self.nitrO.apps[appID]
    loadAspects(app,aspectFacilities)
    self.window.write(app.language.
    getClassesFromSymboltable(app))
  else:
    nitrO.shell.write("The application
    named \""+app+"\"must be star-
    ted.\n")
nitrO.apps["Show Classes Aspect"]._class
_.weave=weave
write ("Routine installed as the\"run\
"method of\"Show Classes Aspect\"applica-
tion.\n")
#>
```

The language neutrality is achieved by following the next steps:

1. The "aspectFacilities" module implements facilities by following a naming convention: their names must be composed of the language identifier, an underscore and the routine's name—e.g. "MetaPython_getClassesFromSymbolTable" in the example above.
2. Every time the "loadAspects" function is called, it analyses every facility developed in the module by means of introspection. This function is called passing an application object as a parameter. In case the application language would be the same as the beginning of the routine's identifier being analyzed, this function will be inserted set as a method of the application object—using runtime structural reflection. This dynamic load offers the possibility to enhance the number of existing facilities while the system is running.

The following code shows the "loadAspects" function implementation:

```
def loadAspects(app,module):
  language=app.language
  try:
    language.aspectRoutinesLoaded
    return # Already loaded
  except:
    pass
  count=0
  for i in dir(module):
    if i .find("_")!=-1 and i.count("_")
    ==1:
     l=i.split("_")
     if  language.name==l[0]:#  Same
     language
      count=count+1
      exec("language."+l[1]+"=module."+i)
  if not(count):
    raise language.name+"must implement
    aspect routines"
  else:
    language.aspectRoutinesLoaded=1
```

The resulting framework follows the "Template Method" design pattern (Gamma et al., 1995) in a runtime reflective way, offering the programmer language-neutrality of every aspect facility (different examples of these facilities are "getClassesFromSymbolTable", "injectCodeIntoMethodCall" and "deleteCodeFromMethodCall"). These facilities offer dynamic aspect adaptation modifying language semantics at runtime. Introspection and structural reflection features can be used to make aspect development easier—like two examples above.

Our non-restrictive reflective platform gives us the opportunity to adapt running applications, even if the aspect is implemented after the application has been executed; using introspection and reflection, aspects can be dynamically woven as well as unwoven. Neither join-point definitions, nor MOP primitives, restrict the set of features that can be adapted.

As an example of using our reflective aspect framework we have developed a dynamic user authentication aspect. Once the "Bank App" program has been started, we may implement an "Authentication Aspect" that would dynamically restrict in some way system's method-invocation semantics. In our example, we authenticate anyone who sends a "withdraw" message to every "Account" object, verifying if that user has permission to make the withdrawal. This is a resume of that implementation:

```
Application="Authentication Aspect"
 ...
def weaveAuthentication(self,appID,className,methodName):
  "Language-independent authentication-aspect weaving routine"
  from aspectFacilities import *
  import aspectFacilities
  app=self.nitrO.apps[appID]
  loadAspects(app,aspectFacilities)
  lang=app.language
  if not(lang.hasMethodCallThisCode(app,lang.nameOfMethodBeingInvokedSemantics)):
     lang.injectCodeIntoMethodCall(app,lang.nameOfMethodBeingInvokedSemantics)
  if not(lang.hasMethodCallThisCode(app,lang.classNameOfMethodBeingInvokedSemantics)):
     lang.injectCodeIntoMethodCall(app,lang.classNameOfMethodBeingInvokedSemantics)
  if not(lang.hasMethodCallThisCode(app,lang.implicitObjectInMethodInvocationSemantics)):
     lang.injectCodeIntoMethodCall(app,lang.implicitObjectInMethodInvocationSemantics)
  app.LoginWindow=self.LoginWindow
  # Sample Authentication: login same as 'user' attribute
  authCode = "if nodes[0].className == '"+className+"'and nodes[0].methodName == '"+methodName+"':"
  authCode=authCode+"""
  application.loginWindow=application.LoginWindow(application.window.master,'Authentication')
  if nodes[0].object.user!=application.loginWindow.login:
     raise 'User not authenticated!'
"""
  if not(lang.hasMethodCallThisCode(app,authCode)):
     lang.injectCodeIntoMethodCall(app,authCode,3)
def unweaveAuthentication(self,app,className,methodName):
  "Language-independent authentication-aspect unweaving routine"
  from aspectFacilities import *
  import aspectFacilities
  app=self.nitrO.apps[appID]
  loadAspects(app,aspectFacilities)
  lang=app.language
  if lang.hasMethodCallThisCode(app,lang.nameOfMethodBeingInvokedSemantics):
     lang.deleteCodeFromMethodCall(app,lang.nameOfMethodBeingInvokedSemantics)
  if lang.hasMethodCallThisCode(app,lang.classNameOfMethodBeingInvokedSemantics):
     lang.deleteCodeFromMethodCall(app,lang.classNameOfMethodBeingInvokedSemantics)
  if lang.hasMethodCallThisCode(app,lang.implicitObjectInMethodInvocationSemantics):
     lang.deleteCodeFromMethodCall(app, lang.implicitObjectInMethodInvocationSemantics)
  authCode="if nodes[0].className=='"+className+"'and nodes[0].methodName=='"+methodName+"':"
  authCode=authCode+"""
  application.loginWindow=application.LoginWindow(application.window.master,'Authentication')
  if nodes[0].object.user!=application.loginWindow.login:
     raise 'User not authenticated!'
"""
  if lang.hasMethodCallThisCode(app,authCode):
```

```
[45]     lang.deleteCodeFromMethodCall(app, authCode)
[46] # Set the function as an application method
[47] nitrO.apps["Authentication Aspect"].LoginWindow = LoginWindow
[48] nitrO.apps["Authentication Aspect"].__class__.weave = weaveAuthentication
[49] nitrO.apps["Authentication Aspect"].__class__.unweave = unweaveAuthentication
[50] write("Aspect \" Authentication Aspect\" installed.\ n")
[51] write("Run nitrO.apps[\"Authentication Aspect\"].weave(\"AppName\",\"Class-
     Name\",\"MethodName\") to weave an application.\n")
[52] write("Run nitrO.apps[\"Authentication Aspect\"].unweave(\"AppName\",\"ClassName\",\"Meth-
     odName\")to unweave an application.\n")
[53] write("Closing this window the aspect will be uninstalled.\n")
[54] #>
```

As in previous examples, this code has its own language specification consisting in a simple reify statement which defines a "weave" (lines 2–24) and "unveawe" function (lines 25–45) and sets them as two aspect methods (lines 48and 49). The weave method enhances the message-passing semantics using different aspect

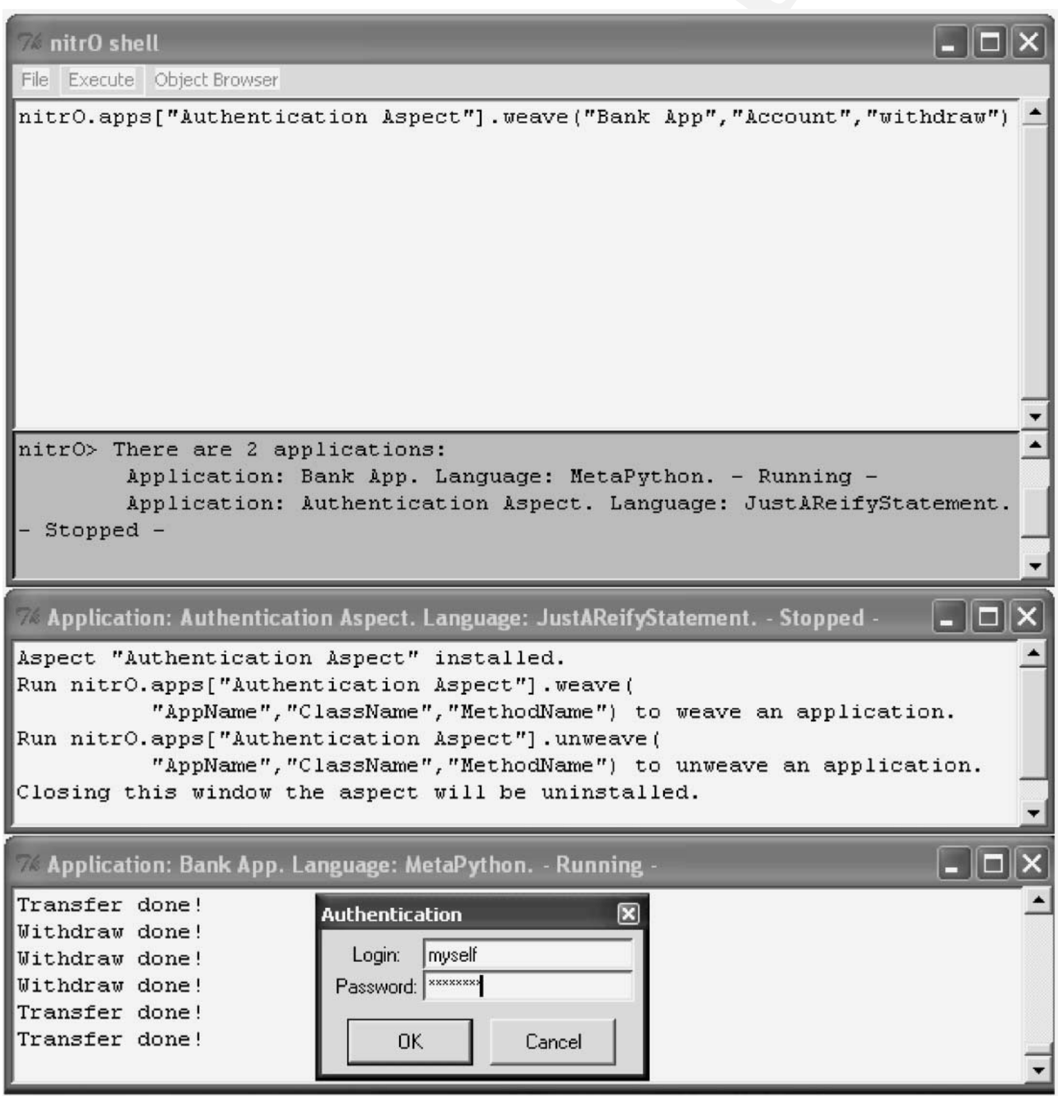

Fig. 7. Dynamic aspect weaving in the nitrO system.

framework facilities. The new semantics checks if the class and method names are the same as the parameters; in that case, a window asking for the user's identity will be prompted. Following this modularization scheme, in which aspect code is decoupled from its join-point identification, our platform can be used as a highly reusable aspect-development system.

We can use this aspect to dynamically set an authentication system to any running application. If the result of weaving an application is not the one the user is expecting—or it is no more needed—, it can be dynamically suppressed by means of the "unweave" aspect method. Note that any dynamically configurable authentication schema based on runtime-emerging contexts, as well as any kind of aspect runtime adaptation, could be performed with this framework.

The running application windows are shown in Fig. 7. After having executed the "Bank App" program (the lower graphic window), we might need to install a security mechanism to make withdrawals. Running the "Authentication Aspect" program (the window in the middle), the aspect will be installed in the system. If we want to dynamically assign the aspect to the running application, we just have to execute the next statement in the nitrO shell (upper window):

```
nitrO.apps["Authentication   Aspect"].
weave("Bank App","Account","withdraw")
```

As shown in the center window of Fig. 7, when the application is about to invoke the "withdraw" method, a login window is shown because of the weaving process just performed. If the user is authenticated, the method will be executed (displaying the corresponding "Withdraw done!" message); in other case, the application throws a "User not authenticated!" exception. The code presented simply authenticates users by comparing their logins with the "user" object's attribute. Obviously, real applications would verify user's identity following different techniques, but this clear example shows how aspects can easily interact with applications. This interaction is straightforward in the nitrO platform because Python is always the unique system's meta-language.

In this sample scenario, the application functional code has not been modified: we customize its language semantics in the weaving process by means of computational reflection. We use Python as a meta-language instead of defining application join-points or MOP-based frameworks; so, our system does not restrict the range of points where an aspect-advice call can be placed.

7.1. Dynamic adaptation of advanced aspects

We are currently developing advanced dynamic aspects over the nitrO platform applied to Java and Python language specifications. An example is a group of aspects that gives an application the ability to be woven at runtime; they make specific objects persist by means of different indexing mechanisms and various levels of persistence (Ortin et al., 1999).

Following the principle of separation of concerns, these Java aspects separate the application functional code from its persistence concerns. Dynamically, based on different runtime-emerging conditions (such as system load, time of the day, or a momentary requirement of faster application execution), different levels of persistence can be assigned to runtime objects, neither having to modify its functional code nor needing to stop its execution. Our system has been designed to be adaptable to different indexing mechanisms (Single Class, CH-Tree and Nested Index) and updating frequencies (creation and deletion of objects, modification of object's state and at regular intervals of time).

Persistence aspects are being developed using different levels of reflection:

1. Introspection is used to obtain existing objects and classes as well as all of their fields and methods. This information is dynamically serialized and saved on disk by using system introspection.
2. Structural reflection is employed to dynamically create, modify and erase existing objects, classes, fields and methods. The need to perform these operations in our persistence system emerges at runtime—this is the reason why the use of reflection is essential.
3. Computational reflection is the key concept employed to link the application functional code with the persistence aspect routines. At present, we customize object creation, object deletion and method invocation semantics.

8. Runtime performance

The main disadvantage of dynamic weaving is runtime performance (Böllert, 1999). The process of adapting an application at runtime, as well as the use of reflection, induces a certain overhead at the execution of an application (Popovici et al., 2001).

Although there are aspects that will benefit from the use of dynamic weaving, this is not needed in many cases. If this situation occurs, static weaving should be used in order to avoid performance penalties. In our platform, the weaving process could also be done statically the same way it is performed at runtime: modifying language specifications.

While developing aspect-oriented applications, the dynamic adaptation mechanism is preferable because it facilitates incremental weaving and makes application debugging easier. Upon deployment, aspects that do not need to be adapted at runtime should be woven statically for performance reasons.

Another performance limitation of our reflective platform is caused by the interpretation of every programming language. Nowadays, many interpreted languages are commercially employed—e.g., Java (Gosling et al., 1996), Python (Rossum, 2001) or C# (Archer, 2001)—due to optimization techniques such as just-in-time (JIT) compilation or adaptable native-code generation (Hölzle and Ungar, 1994). In the following versions of the nitrO platform, these code generation techniques will be used to optimize the generic-interpreter implementation. As we always translate any language into Python code, a way of speeding up application execution is using the interface of a Python JIT-compiler implementation—such as the exploratory implementation of Python for. NET (Hammond, 2001) that uses the. NET common-language-runtime (CLR) JIT compiler.

9. Conclusions

AOP is focused on the separation of crosscutting application concerns. Aspect-oriented tools create the final application by weaving both the program's functional code and the application's specific aspects. Separating the main code from the specific crosscutting concerns makes application source code not being tangled, achieving ease of creation, debug, maintenance and adaptation of applications to new aspects.

Most AOP tools simply support static weaving, not offering the ability to dynamically adapt or replace application aspects by means of dynamic-weaving techniques. Although many aspects do not need this flexibility, specific ones could benefit from it. The few existing dynamic weavers offer runtime adaptability in a restricted way. They also lack language independence, not offering a system in which adaptability is achieved regardless of the programming language being used.

We have developed the nitrO platform in which a non-restrictive reflective technique has been implemented to overcome the previously mentioned limitations. This platform has been used to develop a language-independent AOP framework that offers dynamic aspect (un)weaving, without any predefined restriction. Applications can be dynamically adapted to unpredicted design-time concerns.

Application concerns are defined at the programming language level, not at the application level. This feature offers aspect reutilization when developing system aspects such as persistence or security (Foote, 1992).

The AOP framework offers a language-independent aspect-development system based on dynamic detection and load of specific language routines. We separate the language-neutral aspect development from the specific-language function implementations that have to be included separately in the framework.

The platform also offers great application interoperability. Any application may inspect, and dynamically modify, any aspect of another program—an application may also adapt itself. Therefore, there is no need to stop an application in order to adapt it at runtime: another one may be used to customize the former.

Finally, no restrictions are imposed by application join-points specification. In most AOP tools, applications must define points where they might be adapted at runtime, by previously specifying their join-points. Others, like PROSE (Popovici et al., 2001), use a MOP that also restricts its adaptability. In nitrO, the whole application is adaptable at runtime: its structure and its programming language's semantics can be inspected and dynamically modified—not needing to previously specify *what* might be customized. Applications can be adapted to new runtime-emerging aspects, unpredictable at design time.

The current platform implementation has performance disadvantages. However, we expect that the employment of a Python JIT compiler in future versions will show common dynamic-weaving performance. The main goal of our first implementation was overcoming the limitations of existing dynamic-weaving tools we have pointed out.

The Python platform source code and some testing applications can be downloaded from `http://www.di.uniovi.es/reflection/lab/prototypes.html#nrrs`.

Acknowledgements

This work was supported in part by the *Laboratory of Object-Oriented Technologies* research group (`http://www.ootlab.uniovi.es`) of the University of Oviedo (Spain).

References

Andersen, A., 1998. A note on reflection in Python 1.5, Distributed Multimedia Research Group, Report. MPG-98-05, Lancaster University, UK.

Archer, T., 2001. Inside C#. Microsoft Press.

Böllert, K., 1999. On weaving aspects. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming.

Chiba, S., Michiaki, T., 1998. A yet another java.lang.class. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Reflective Object Oriented Programming and Systems.

Douence, R., Südholt, M., 1999. The next 700 reflective object-oriented languages, Technical report no.: 99-1-INFO, École des mines de Nantes Dept. Informatique.

Foote, B., 1992. Objects, reflection, and open languages. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Object-Oriented Reflection and Metalevel Architectures.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley.

Gosling, J., Joy, B., Steele, G., 1996. The Java Language Specification. Addison-Wesley.

Hammond, M., 2001. Python for. NET: Lessons learned, Active State Corporation.

Hölzle, U., Ungar, D., 1994. A third-generation SELF implementation: reconciling responsiveness with performance. In: Proceedings of the Object-Oriented Programming Languages, Systems and Applications (OOPSLA) Conference.

Hürsch, W.L., Lopes, C.V., 1995. Separation of Concerns, Technical Report UN-CCS-95-03, Northeastern University, Boston, USA.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. Getting Started with AspectJ, Communications of the ACM, October.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J., 1997. Aspect oriented programming. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), vol. 1241. Lecture Notes in Computer Science, Springer-Verlag.

Kiczales, G., Rivieres, J., Bobrow, D.G., 1992. The Art of Metaobject Protocol. MIT Press.

Kleinöder, J., Golm, M., 1996. MetaJava: an efficient run-time meta architecture for Java™. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Object Orientation in Operating Systems.

Maes, P., 1987. Computational Reflection, PhD. Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels, Belgium.

Matthijs, F., Joosen, W., Vanhaute, B., Robben, B., Verbaten, P., 1997. Aspects should not die. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect-Oriented Programming.

O'Brien, L., 2001. The First Aspect-Oriented Compiler, Software Development Magazine, September.

Ortin, F., Cueva, J.M., 2001. Building a completely adaptable reflective system. In: European Conference on Object Oriented Programming (ECOOP) Workshop on Adaptive Object-Models and Metamodeling Techniques.

Ortin, F., Cueva, J.M., 2002. The nitrO reflective platform. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP), Session on Adaptable Software Architectures.

Ortin, F., Martínez, A.B, Álvarez, D., Cueva, J.M., 1999. An implicit persistence system on an OO database engine using reflection. In: Proceedings of the International Conference on Information Systems Analysis and Synthesis (ISAS), July.

Parnas, D., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12).

Pinto, M., Amor, M., Fuentes, L., Troya, J.M., 2001. Run-time coordination of components: design patterns vs. component and aspect based platforms. In: European Conference on Object-Oriented Programming (ECOOP) Workshop on Advanced Separation of Concerns.

Popovici, A., Gross, Th., Alonso, G., 2001. Dynamic Homogenous AOP with PROSE, Technical Report, Department of Computer Science, ETH Zürich, Switzerland.

Rossum, G., 2001. Python Reference Manual. In: Fred L. Drake Jr. (Ed.), Relesase 2.1.

Smith, B.C., 1984. Reflection and semantics in Lisp. In: Proceedings of ACM Symposium on Principles of Programming Languages.

Tarr, P., Ossher, H., Harrison, W., Sutton, S., 1999. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 1999 International Conference on Software Engineering.

Zinky, J.A., Bakken, D.E., Schantz, R.E., 1997. Architectural support for quality of service for CORBA objects. Theory and Practice of Object Systems 3 (1).

Francisco Ortin (Oviedo, Spain, 1973): Computer Scientist from the Technical School of Computer Science of the University of Oviedo (1994) and Computer Engineer from the Technical School of Computer Science of Gijón (1997). Ph.D. from Computer Science Department of the University of Oviedo (2002) with a Thesis entitled "A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine". Working as a Lecturer at the Computer Science Department of the University of Oviedo, his main research interests are Computational Reflection, Object-Oriented Abstract Machines, Meta-level Systems and Meta-Object Protocols, Aspect-Oriented Programming and any kind of Adaptable Systems. More details can be found in: http://www.di.uniovi.es/~ortin.

Juan Manuel Cueva (Oviedo, Spain, 1958): Mining Engineer from Oviedo Mining Engineers Technical School in 1983 (Oviedo University, Spain). Ph.D. from Madrid Polytechnic University, Spain (1990). From 1985 he is a Professor at the Languages and Computers Systems Area in Oviedo University (Spain). Currently, he also works as the Director of the Technical School of Computer Engineering of Oviedo University. ACM and IEEE voting member. His research interests include Object-Oriented Technology, Language Processors, Human-Computer Interface, Object-Oriented Databases, Web Engineering, Object-Oriented Languages Design, Object-Oriented Programming Methodology, XML, Modeling Software with UML and Geographical Information Systems. More details can be found in: http://www.di.uniovi.es/~cueva/.