# Building a Completely Adaptable Reflective System

Francisco Ortín Soler, Juan Manuel Cueva Lovelle

Campus Llamaquique, c/Calvo Sotelo s/n
33007 Oviedo (Spain)
{ortin, cueva}@pinon.ccu.uniovi.es

**Abstract.** Reflection is one of the main techniques used to develop adaptable systems and, currently, different kinds of reflective systems exist. Compile-time reflection systems provide the ability to customize their language but they are not adaptable at runtime. On the other hand, runtime reflection systems define meta-object protocols to customize the system semantics at runtime. However, these meta-object protocols restrict the way a system may be adapted before its execution, and they do not permit the customization of its language.

Our system implements a non-restrictive reflection mechanism over a virtual machine, in which every feature may be adapted at runtime. No meta-object protocol is used and, therefore, it is not needed to specify previously what may be reflected. With our reflective system, the programming language may be also customized at runtime.

## 1 Introduction

Many reflection techniques have been developed to build adaptable systems. Compile-time reflection achieves adaptable programming languages with efficient performance (e.g., openC++ [1], MPC++ [2] or openJava [3]); however, it lacks adaptability at runtime [3]. Most of the runtime-reflection systems are based on the ability to modify the programming language semantics while the application is running (e.g., message passing mechanism). This adaptability is achieved by implementing a protocol as part of the interpreter; the protocol specifies (and therefore, restricts) the way a program could be modified at runtime.

We have designed a runtime reflective system, in which it is possible to change every feature of the programming language at runtime without any restriction imposed by an interpreter protocol.

Our first step was the implementation of a structural-reflective virtual machine called nitrO Virtual Machine. Its basic object-oriented prototype-based computation model plus its structural reflection feature, permit us to build the system with its own language.

Codified with the nitrOVM programming language, we have developed a language processor tool in order to construct generic interpreters. Specifying lexical, syntactic and semantic rules of a programming language, an interpreter would be generated. At runtime, every program will be able to customize its own language accessing its language specification, nevertheless which language has been used.

The final system achieves complete adaptability at runtime, it may be used with any programming language, and no protocol restrictions exist at runtime.

The rest of this paper is structured as follows. In the next section we briefly describe reflection classifications, its advantages and drawbacks. In section 3 we present the

nitrO Virtual Machine. The language processor tool and its benefits are described in sections 4 and 5. Then we comment related work, and our conclusions are shown on section 7.

# 2 Categorizing Reflection

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

- *Introspection*: The system 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, objets, methods and fields at runtime [4].
- *Structural Reflection*: The system structure can be dynamically modified. An example of this kind of reflection is the addition of object's fields.
- *Computational* (*Behavioral*) *Reflection*: The system semantics (behavior) can be modified. In the standard Java API v.1.3, the class `java.lang.reflect.Proxy` has been added [5]; it can be used to modify the dispatching method mechanism, being handled by a proxy object.
- A reflective programming language can be capable of changing itself, i.e. changing its own lexical or syntactical specification; that is what we call *Linguistic Reflection*. As an example, with OpenJava reflective language, the own language can be enhanced to be adapted to specific design patterns [6].

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 [3]). The benefits of this system are runtime performance and the ability to adapt its own language (i.e., *linguistic reflection*).
- *Runtime Reflection*: The system may be adapted at runtime, once it has been created and run (e.g., metaXa, formerly called MetaJava [7]). These systems have greater adaptability but performance penalties. Computational reflective systems are commonly implemented by using *runtime reflection* by they lack *linguistic reflection* capabilities.

Our system, nitrO, achieves *computational* and *linguistic* reflection at *runtime*. Moreover, our reflection technique implementation is more flexible than common runtime reflective systems –as we will explain in the next paragraph. Performance drawbacks are not being considered in our first prototypes.

## 2.1 Meta-object Protocols Restrictions

Most runtime reflective systems are based on Meta-Object Protocols (MOPs); a MOP specifies the implementation of a reflective object-model [8]. An application is implemented by means of a programming language (base level). A program's meta-level is the implementation of the computational object model supported by the programming language. Therefore, a MOP specifies the way a base-level application may access its meta-level in order to adapt its behavior at runtime.

The way a MOP is defined restricts the amount of features that may be customized. If we do not consider a system feature to be adaptable by the MOP, this program attribute will not be able to be customized once the application will be running.

nitrO runtime reflection mechanism is based on a meta-language specification. The way the base level access to the meta-level (reification) is not defined by a MOP; it is specified by another language (meta-language). The meta-language is capable to adapt the structure, behavior and linguistic features of the base level system at runtime. Its design will be specified in section 4.

# 3 nitrO Virtual Machine

Achieving computational reflection by an interpreter is easier than generating native code to a specific platform (e.g., the difference between the `java.lang.reflect` Java package [4] and the RTTI C++ mechanism [9]). In the case of interpretation, the program and the own interpreter both run on the operating system process, achieving interoperability between them in a simpler way. Common interpreted computational-reflective programming language scenarios are:

- An interpreter of a language capable to modify its own semantics (e.g., CLOS [8]).
- A virtual machine that executes portable code over different platforms and is capable to change its own behavior (e.g. metaXa [7]).

Both scenarios are based on the use of meta-object protocols and, as we have mentioned on the section above, they have certain limitations.

The use of a virtual machine has many advantages in order to build a flexible computation platform:

- Portable code to any platform.
- Programming language independence (not designed to be used by just one language).
- Unique object and computation model defined by the virtual machine, to be used in the whole system.
- Easy development of distributed applications over different virtual machines.
- Code mobility and distribution over a network.

Therefore, we have implemented our system over a virtual machine called nitrO Virtual Machine (nitrOVM). These are its main features:

**Prototype-based object model**: Our basic abstraction is the object. We use a prototype-based object model in which classes do not exist.

- The virtual machine uses a simpler object model eliminating the class abstraction. Trait objects may be used to describe behavior of a group of objects [10].
- There is no representation loss by using a prototype-based object model instead of using a class-based model [11].
- Different object-oriented languages can be easily translated into a prototype-based object-oriented language [12].
- In a class-based model, in order to change one object's structure, the whole class must be modified, but what happen to other class' objects? MetaXa creates a new class, called shadow class, just for this object, making the object model more complex and difficult to implement [13]. In a prototype-based model this is easily achieve by means of object cloning and dynamic inheritance (delegation) [10].

**Structural reflection**. Main features:

- An object is defined as a collection of references to other objects.
- There are two primitive objects: the `nil` object and string objects.

- Every object offers structural-reflective operations (delete, insert and iteration of every element of the reference collection). A dynamic inheritance mechanism exists and the root object is `nil` –the one who offers the structural-reflective operations mentioned.
- Behavior is expressed by means of string objects; these may be dynamically created, manipulated and evaluated (reflected) with the `()` operator. A method is defined as an evaluable string that is a member of the object.

**Extensibility**: The virtual machine computational model has been designed to be very reduced. By means of structural reflection, a programming environment has been developed over the basic virtual machine language (thus, it is platform-independent and portable code). This programming environment achieves a higher abstraction level: adaptable persistence, distribution and thread scheduling systems have been developed in this programming environment.

**Applications interoperability**: Just one virtual machine exists for every physical computer. Opposite to Java Virtual Machine [14], different applications in the same physical computer use the same nitrOVM. The main benefit is that every application is capable to access every object in the virtual machine (not just its own objects) by using structural reflection.

More information related to the virtual machine design and implementation as well as its programming language and project status may be found in our web site [15].

## 4 Non-restrictive Computational Reflection System

Most common compiler and interpreter construction tools (from classic lex and yacc to modern JavaCC [16] and AntLR [17]) have the same development process, shown in figure 1:

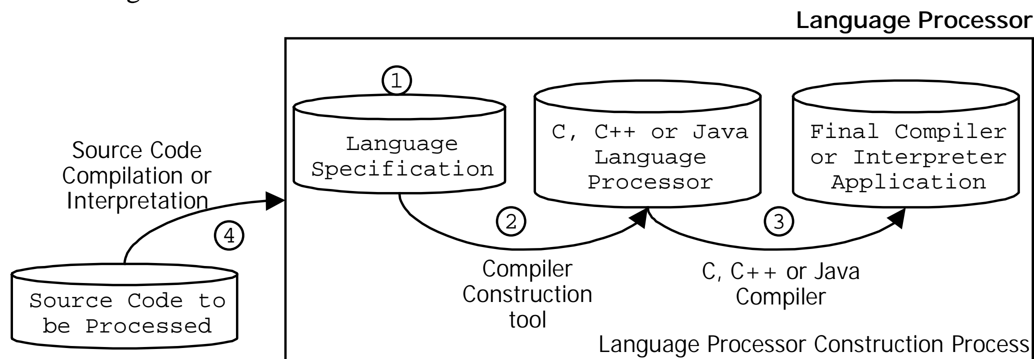


**Fig. 1.** Common compiler or interpreter development process.

1. The language specification is detailed using the tool's language.
2. The language specification is preprocessed and a C, C++ or Java application is generated.
3. Once the application has been generated, it is compiled to obtain the language processor.
4. With the compiler or interpreter created, we can process source code of the desired language.

If we want a language feature to be modified, the whole process has to be repeated. Therefore, by using this scheme, no dynamic modification may be done to the language being processed.

"Generic Interpreter" [18] interpreter construction tool is an exception: language specification may be dynamically changed. However, the changes must be codified and compiled previously to the interpreter execution.

Our language processor is defined as a Structural-Reflective Generic Interpreter (SRGI). Lexical and syntactic specifications are represented by objects meaning free-context grammar rules [19]. The object represents the left side of the rule and the right side is represented by the member collection the object has. Rules may be created, analyzed and modified dynamically by using virtual machine structural reflection.

**Free-Context Grammar Rule:**

```
<Statements> → <Statement> ; <Statements>
```

**Object-Oriented Representation:**

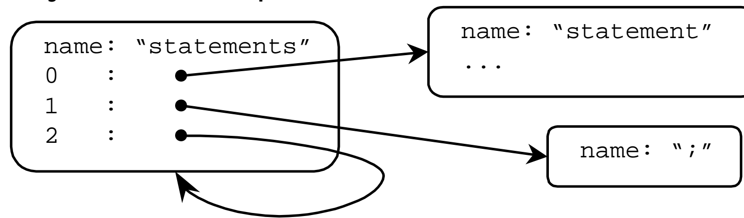


**Fig. 2.** Object's structure representing a free-context grammar rule.

Semantic specification associated to syntactic rules is described as string objects. These can be easily evaluated by the virtual machine, using the `()` operator.

The SRGI Engine has been designed to use a backtracking algorithm. Once the language to be interpreted is specified, the SRGI Engine starts processing it following a top-down scheme [19]. The result is a two-level interpreter tower [20]: first the virtual machine and second the language being interpreted by the SRGI Engine.

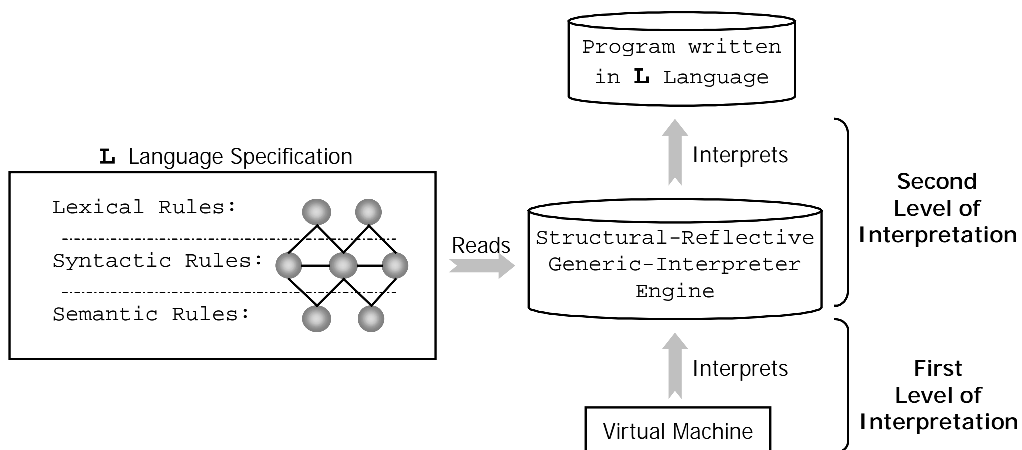


**Fig. 3.** Structure of the two levels in generic interpretation.

The SRGI can interpret any language –we will generically call it **L**– based on its specification, and it would be always capable to interpret one language without specifying it: the nitrOVM language. Using the reserved word `reflect`, nitrOVM code may be written. The SRGI Engine takes the code as a string object and evaluates it with the `()` operator. Thus, the first level (nitrOVM), instead of the second (SRGI), evaluates the code written inside the `reflect` statement, so one level of interpretation has been jumped (shown in figure 4).

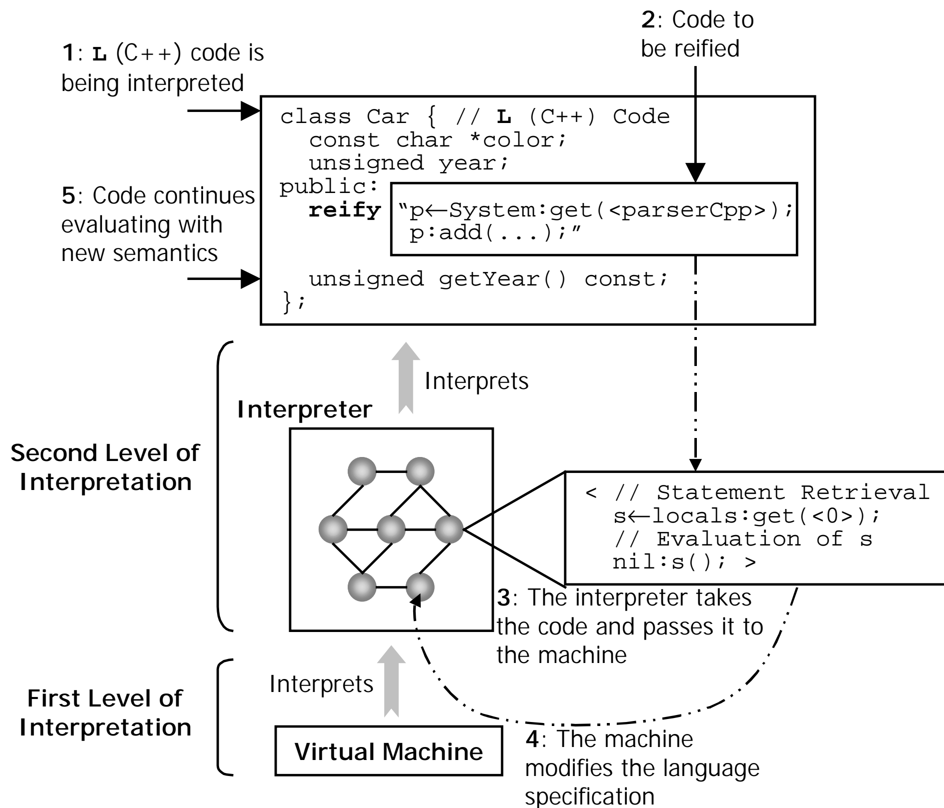


**Fig. 4.** Achieving a jump in the two-level tower of interpreters.

If the reflected nitrOVM code modifies the `L` code specification by means of nitroVM structural reflection, what we achieve is a non-restrictive computational reflection mechanism. With this scheme, the nitrOVM language becomes a meta-language to specify, and dynamically modify, the language that would be interpreting; no previous MOP specifying *what* may be changed has to be defined.

# 5 System Benefits

Our flexible platform offers the following advantages:

- The system is platform independent: it has been developed over the virtual machine language.
- The system may be programmed by means of any programming language: the SRGI can interpret any language; different language specifications can be saved by the programming environment persistence system.
- The whole system is adaptable at runtime: any feature of the system may be changed by means of the `reflect` statement, and there are no previous restrictions imposed by any protocol.
- Any feature may be reflected: introspection, structural reflection, computational reflection and "linguistic" reflection are achieved.
- Application interoperability: any program run by the nitrOVM may access, and reflectively modify, any other program being executed by the same virtual machine. Therefore, there is no need to stop an application if we want to adapt it at runtime: another application may be used to adapt the former.

The result is a universal computation platform that may be used to develop or test any reflective or adaptable environment (e.g., fault-tolerant systems, adaptable operating systems, knowledge base systems, etc.) without the necessity to modify the virtual machine source code.

The main disadvantage is performance penalties. Future work will be studying and implementing optimization techniques as just in time compilation or native code generation.

# 6 Related Work

One of the most advanced computational-reflection systems based on the development of a virtual machine is MetaXa (formerly called MetaJava) [7]. It is based on a runtime meta-object protocol over a modification of the Java virtual machine [14]. In our research group, with have also defined a MOP over an object-oriented abstract machine (our first prototype) in order to build an integral object-oriented system [21]. We realized that the abstract machine should be modified every time a new feature would be added to our MOP -as we have mentioned in section 2.1; therefore, we discarded MOP-based systems.

Another way to develop a language-independent flexible platform is designing the virtual machine in a modular way; this technique has been used to build the "Virtual Virtual Machine" [22]. Different parts of a platform are selected to make them adaptable (e.g., thread scheduling or security policies), and many languages can be processed by this platform. However, only those limited features selected in the platform architecture would be adapted at runtime.

# 7 Conclusions

Most systems that offer computational reflection at runtime are based on the use of a meta-object protocol (MOP). MOPs give a system the ability to customize at runtime, but *what* may be adapted must be previously specified by the protocol. Different approaches modifying the MOP are commonly needed to make the system adaptable to a new characteristic. Moreover, this kind of system lacks the ability to modify its own language ("linguistic" reflection) and the cross-customization between different applications cannot be achieved either.

Our computation system's root is an object-oriented prototype-based virtual machine endowed with structural reflection. Over this virtual machine, we have designed a structural-based language specification that represents lexical, syntactic and semantic free-context grammar rules. An application (engine) executes the language rules following a top-down scheme, achieving any programming language interpretation.

The interpreter engine is capable to obtain virtual machine code (using the `reflect` statement) and make it evaluate by the virtual machine. Modifying the language specification implies computational and "linguistic" reflection without any restriction. A real meta-level jump is obtained and no changes to the virtual machine implementation have to be done.

The final system is a computation platform that might be programmed by using any language, it is completely adaptable, and it has a great level of application interoperability; therefore, it can be used to create or test highly adaptable environments.

## References


1 Shigeru Chiba. “A Metaobject Protocol for C++”. *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. No. 10 in SIGPLAN Notices vol. 30, ACM, 1995.

2 Y. Ishikawa. “Meta-Level Architecture for Extendable C++”. Technical Report TR-94024. Tsukuba Research Center. Japan, 1995.

3 Shigeru Chiba and Tatsubori Michiaki. “A Yet Another java.lang.Class”. *ECOOP’98 Workshop on Reflective Object Oriented Programming and Systems*. Brussels, Belgium. July, 1998.

4 “The Java Core Reflection API and Specification”. Sun Microsystems Computer Corporation. January, 1997.

5 “Dynamic Proxy Classes”. Sun Microsystems Computer Corporation. 1997.

6 Michiake Tatsubori and Shigeru Chiba. “Programming Support of Design Patterns with Compile-time Reflection”. *OOPSLA’98 Workshop on Reflective Programming*”. Vancouver (Canada). October, 1998.

7 Jügen Kleinöder and Michael Golm. “MetaJava: An Efficient Run-Time Meta Architecture for Java™”. *International Workshop on Object Orientation in Operating Systems, IWOOOS’96.* Seattle, Washington. October, 1996.

8 Gregor Kiczales, J. Des Rivieres and D. G. Bobrow. “The Art of Metaobject Protocol”. MIT Press, 1992.

9 Bjarne Stroustrup. “The C++ Programming Language”. Addison Wesley Editorial, 3rd Edition. October, 1998.

10 David Ungar and R. B. Smith. “SELF: The Power of Simplicity”. *Object Oriented Programming, Systems, Languages and Applications OOPSLA*. 1987.

11 David Ungar, Craig Chambers, Bay-Wey Chang and Urs Hölzle. “Organizing Programs without Classes”. *Lisp and Symbolic Computation*. 1991.

12 Mario Wolczko, Ole Agesen and Davied Ungar. “Towards a Universal Implementation Substrate for Object-Oriented Languages”. Sun Microsystems Laboratories.December, 1996.

13 Michael Golm and Jügen Kleinöder. “MetaJava – A Platform for Adaptable Operating-Systems Mechanisms” ECOOP Workshop on Object-Orientation and Operating Systems. Jyväskylä, Finland. June, 1997.

14 “The Java Virtual Machine Specification”. Release 1.0 Beta. Draft. Sun Microsystems Computer Corporation. August, 1995.

15 http://www.di.uniovi.es/reflection/lab/. "Computational Reflection Research Group". University of Oviedo. Spain.

16 "Java Compiler Compiler (JavaCC) -The Java Parser Generator". Sun Microsystems Computer Corporation. 2000.

17 Terence Parr. Antlr Reference Manual. Magelang Institute. January 2000.

18 Craig A. Rich "Generic Interpreter 0.9" http://www.csupomona.edu/~carich/gi/. 2000.

19 Alfred V. Aho, Ravi Sheti and Jeffrey D. Ullman. "Compilers: Principles, Techniques and Tools". Addison-Wesley. November, 1985.

20 B.C. Smith. “Reflection and Semantics in Procedural Language”. MIT-LCS-TR-272. Massachusetts Institute of Technology. 1982.

21 Lourdes Tajes, Fernando Álvarez, María Ángeles Díaz, Juan Manuel Cueva. "Reflection for Extending OO Systems with Distribution Capabilities". *Simposio Español de Infor-*

*mática Distribuida SEID* Ourense (Spain). September, 2000.

22 Bertil Folliot, Ian Piumarta and Fabio Riccardi. "A Dynamically Configurable, Multi-Language Execution Platform". 8th ACM SIGOPS European Workshop. September, 1998.