Non-Restrictive Computational Reflection

Francisco Ortin Juan Manuel Cueva
University of Oviedo
Calvo Sotelo s/n, 33005, Oviedo, Spain
Computer Science Department

***Abstract**-- Adaptable software systems and architectures give the programmer the ability to create applications that might customize themselves to runtime-emerging requirements. Computational reflection is a programming language technique that is commonly used to achieve the development of this kind of systems. Most runtime reflective systems use meta-object protocols (MOPs). However, MOPs restrict the amount of features an application can customize, and the way they can express its own adaptation. Furthermore, this kind of systems uses a fixed programming language: they develop an interpreter, not a whole language-independent platform.*

What we present in this paper a non-restrictive reflective platform, called nitrO, that achieves a real computational-environment jump, making every application and language feature adaptable at runtime –without any previously defined restriction. Moreover, the platform has been built using a generic interpreter, in which the reflection mechanism is independent of the language selected by the programmer. Different applications may dynamically adapt each other, regardless of the programming language they use.

Keywords*:* reflection, computational-environment jump, generic interpreter, separation of concerns.

1 Introduction

Adaptability has become an important feature in modern computing systems, languages and software engineering methods. Different techniques are emerging in order to build adaptable computing systems and software engineering methods. Two examples in the software engineering field are aspect-oriented programming (AOP) [1] and multi-dimensional separation of concerns [2]. They distinguish functional code from application crosscutting concerns, creating the final application by *weaving* the program and its specific aspects. Most of them lack runtime adaptability, simply offering design-time adaptation.

Reflection is a programming language technique that achieves dynamic 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 (Meta-Object Protocol, MOP [3]) as part of the language interpreter that specifies –and therefore, restricts– the way a program can be modified at runtime. As we will explain, other common MOP-based system limitations are their language dependence and their restrictions expressing system's features modification.

What we present here is a non-restrictive reflection technique that we use in the nitrO reflective platform [4]. In nitrO, it is possible to change every feature of its programming languages and applications at runtime, without any kind of restriction imposed by an interpreter protocol. Any programming language can be used, and every application is capable of adapting another one's features, no matter whether they use the same programming language or not.

By using our system, it is possible to develop applications that may be adapted to unpredictable design-time requirements, changing its own structure and behavior at runtime, regardless of which programming language has been used.

The rest of this paper is structured as follows. In the next section we briefly describe two reflection classifications and meta-object protocol systems; we also present the Python programming language and its reflective features. Section 3 introduces our system architecture and its design is presented in section 4. How applications and programming languages are represented is described in section 5 and dynamic-adaptation sample code is shown in the following section. We

summarize our system's benefits and performance limitations in section 7, and section 8 presents the final conclusions.

2 Classifying Reflection

The two main criteria used to classify reflective systems are *when* reflection takes place and *what* system's features can be reflected. Depending on *when* reflection might take place:

- *Compile-time Reflection*: The system customization takes place at compile-time (e.g., OpenJava [5]). The main benefits of this kind of systems are runtime performance and the ability to adapt its own language. Many aspect-oriented tools use this technique.
- *Runtime Reflection*: The system can be adapted at runtime once it has been created and run (e.g., metaXa, formerly called MetaJava [6]). These systems offer greater adaptability that compile-time ones, by paying performance penalties.

If we take *what* can 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, its `java.lang.reflect` package gives the programmer the capability to get information about classes, objects, methods and fields at runtime.
- *Structural Reflection*: The system structure can be dynamically 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, in the MetaXa system [6] the message-passing mechanism can be customized at runtime by the program.

2.1 Meta-Object Protocols Restrictions

Most runtime computational-reflective systems are based on Meta-Object Protocols (MOPs). A MOP specifies the implementation of a reflective object-model [7]. 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 at the interpreter execution environment. Therefore, a MOP specifies the way a base-level application may access its meta-level in order to adapt its behavior and structure at runtime.

As shown in Figure 1, the implementation of different meta-objects can be used to override the system's semantics. For example, in MetaXa [6], we can implement the class `Trace` inheriting from the class `MetaObject` (offered by the language as part of the MOP), overriding the `eventMethodEnter` method. Its instances are meta-objects that can be attached to user objects. Every time a message is passed to these user objects, the `eventMethodEnter` method of its attached meta-objects will be called –showing a trace message and, therefore, customizing its message-passing semantics.

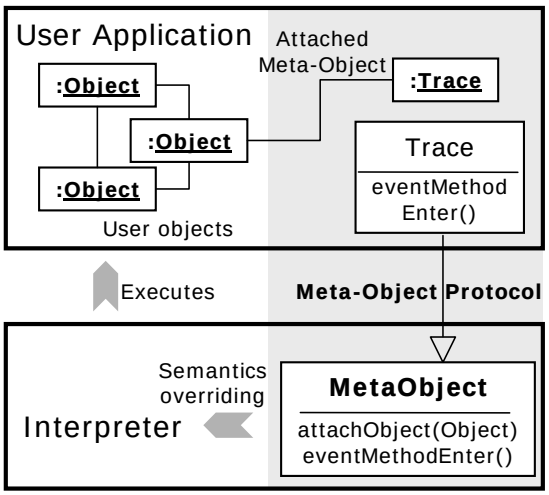

Fig. 1. MOP-based system architecture.

This Meta-Object Protocol reflective technique has different drawbacks:

1. The way a MOP is defined restricts the amount of features that might be customized [8]. 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 is running. In our example, if we want to adapt the way objects are created and the MOP does not offer this possibility, we must stop the program execution and modify the MOP implementation.
2. Changing the Meta-Object Protocol in order to achieve higher adaptability means different interpreter and language versions and, therefore, could make the previous existing code been deprecated.
3. The way a semantic feature can be customized has expressiveness restrictions. Object's behavior may be adapted by attaching a meta-object to him. This meta-object may just express the way it would modify its semantics by overriding its super-class' methods –the interpreter will call this new method every

time a message is passed to the object. The use of a meta-language would be a richer mechanism to express the way an application may be adapted.

4. Finally, MOP-based systems are language-dependent. Meta-level and base-level programming languages are always the same; they do not offer runtime adaptability in a language-independent way.

Our nitrO runtime reflection mechanism is based on the use of a meta-language. The base-level access the meta-level (reification) by means of 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 –whatever the programming language has been used. Its design will be specified in section 4.

2.2 Python's Reflective Capabilities

We have selected the Python programming language [9] to develop our system because of its reflective capabilities [10]:

- Introspection. At runtime, the programmer may inspect any object, its attributes, class and inheritance graph. It may also be inspected the application's dynamic symbol table: the existing modules, classes, objects and variables at runtime.
- 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.

3 System Architecture

The theoretical definition of reflection uses the notion of a reflective tower of interpreters [11]: 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.

Related to the preceding definition, if an application would be able to access its interpreter at runtime, it would be able to inspect the existing system objects (introspection), modify its structure (structural reflection) and customize its language semantics (computational reflection).

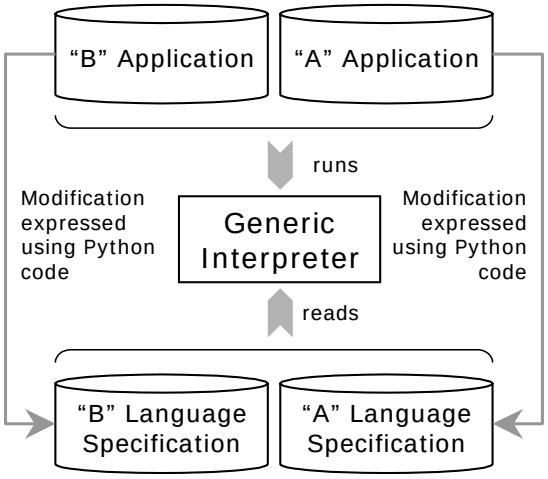

Fig. 2. System architecture.

However, this mechanism is complicate to implement. Interpreters commonly have complex structures representing different functionality like parsing mechanism, semantics interpretation, and runtime user-application representation. For instance, if an application modifies by error the parsing mechanism, it would produce unexpected results.

What we have developed is a generic interpreter (Figure 2) that separates the structures accessible by the base-level from the fixed mechanism that should never be modified. This generic interpreter is language-independent: its inputs are both the user application and the language specification; it is capable of interpreting any programming language by previously reading its specification.

At runtime, any application may access language specifications by using the whole expressiveness of the Python programming language; there are no previously specified restrictions imposed by a protocol –any feature can be adapted. Changes to language specifications are automatically reflected on the application execution because the generic interpreter relies on the language specification while the application is running.

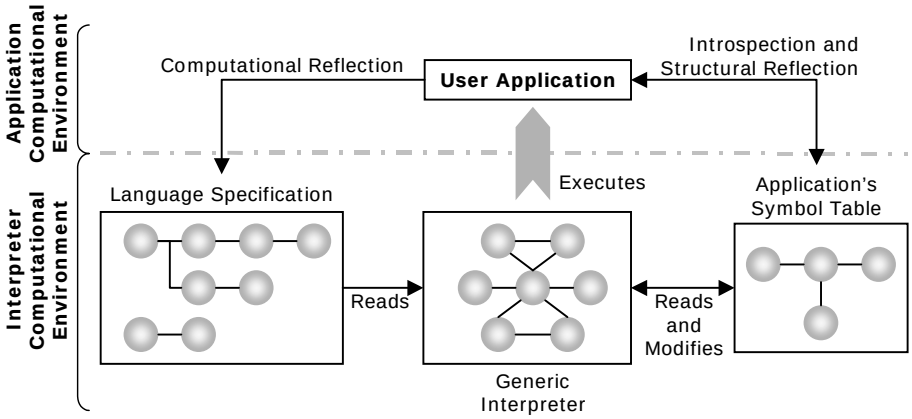

Fig. 3. Language specification and symbol table modification.

4 System Design

In Figure 3 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 application's runtime symbol table: variables, objects and classes created by the user.

Any running 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 reflection capabilities, implies structural reflection of any running application.
3. Adapting the language semantics located in the language specification, the running application may customize its behavior achieving computational reflection.

The main question of this design is how the application computational environment may access and modify the interpreter computational environment –i.e., how a user application may access different language specifications and application's symbol-tables.

Every language in our system includes the

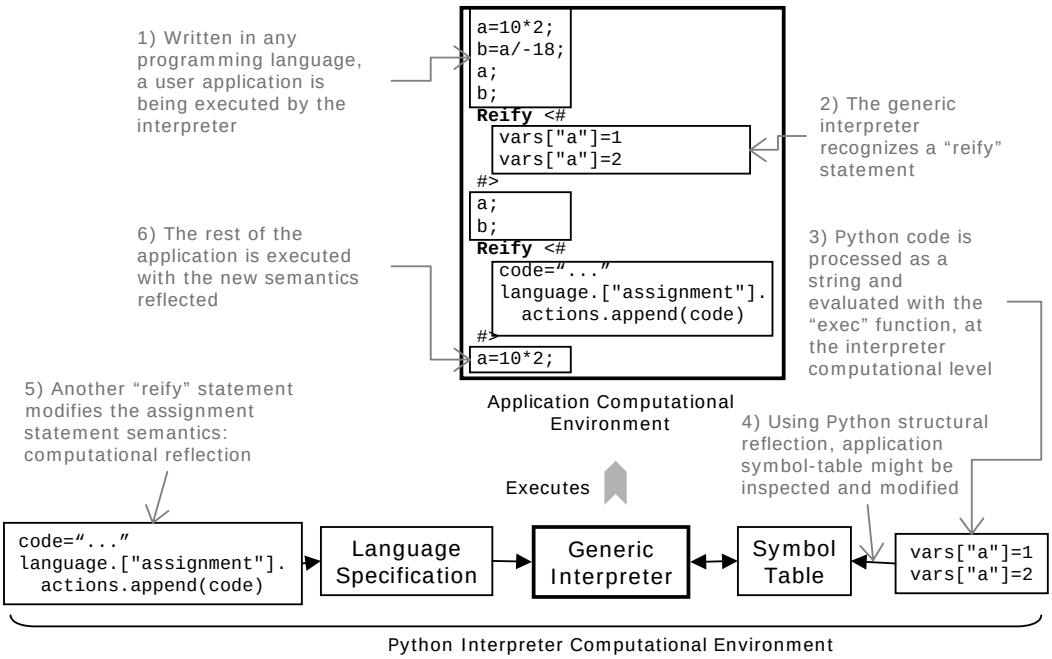

Fig. 4. Achieving a real computational-environment jump.

`reify` statement; the generic interpreter automatically recognizes it, no matter the language being used. Inside a `reify` statement Python code can be written. This Python code will not be processed as the rest of the application code: independently of the programming language selected, every time the interpreter recognizes a `reify` statement, its Python code will be taken and evaluated invoking the `exec` function. This code, using Python structural reflection, may access and modify application's symbol-tables and language specifications. This scheme is shown in Figure 4.

The code written inside a `reify` statement is evaluated in the interpreter computing environment, not in the application computing environment –the place where it was written. So, Python becomes a meta-language to specify, and dynamically modify, any language and application that would be running in our system. There is no need to specify a MOP that would previously restrict which language 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.

Looking for good performance, MOP-based systems simulate the computational-environment jump by offering meta-objects to the programmer; these are executed in the application environment, not at the interpreter level. That is the reason why they lacks features pointed in section 2.1.

5 Language and Application Representation

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 language specification files. Their lexical (`Scanner` section) and syntactic (`Parser` section) features are expressed by means of context-free grammar rules; their semantics, by means of Python code placed at the end of each rule (between `<#` and `#>` characters).

We have specified the Python programming language and some domain-specific languages [12]. Currently we are specifying Java and Jscript languages. Correctness verification (e.g., type checking) is expressed inside the semantic actions using Python code.

What we present here is an example of a "VerySimple" language definition without any semantic correctness verification:

```
Language = VerySimple

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
          |
          ;
  "Character Token"
    char -> "a"|"b"|"c"|"d"|"e"|"f"|
            "g"|"h"|"i"|"j"|"k"|"l"|"m"|
            "n"|"o"|"p"|"q"|"r"|"s"|"t"|
            "u"|"w"|"x"|"y"|"z"
          ;
  "Character or Digit Token"
    charOrDigit -> char | digit
          ;
  "ID Token"
    ID -> char moreCharsOrDigits
          ;
  "Zero or more chars or digits token"
    moreCharsOrDigits -> charOrDigit
                         moreCharsOrDigits
          |
          ;
  "SEMICOLON Token"    SEMICOLON -> ";"
          ;
  "ASSIGN token"    ASSIGN -> "="
          ;
}

Parser = {
  "Initial Context-Free Rule"
    S -> statement moreStatements SEMICOLON <#
global vars
vars={}
nodes[1].execute()
nodes[2].execute()
#>
          ;
  "Zero or more Statements"
    moreStatements -> SEMICOLON statement
moreStatements <#
nodes[2].execute()
nodes[3].execute()
#>
          |
          ;
  "Statement"
    statement -> _REIFY_ <#
nodes[1].execute()
#>
          | assignment <#
nodes[1].execute()
#>
          | expression <#
nodes[1].execute()
```

```
write("Expression value: "+
  str(nodes[1].value)+".\n")
#>
          ;
  "Assignment Statement"
    assignment -> ID ASSIGN expression <#
nodes[3].execute()
vars[nodes[1].text]= nodes[3].value
#>
          ;
  "Binary Expr. Factor"
    expression -> ID <#
nodes[0].value=vars[nodes[1].text]
#>
          | NUMBER <#
nodes[0].value=int(nodes[1].text)
#>
          ;
}

Skip = {"\t";  "\n"; " ";}

NotSkip = {  }
```

The `_REIFY_` reserved word indicates where a `reify` statement might be syntactically placed. `Skip` and `NotSkip` sections tells the interpreter which tokens has to be automatically ignored and which ones should be appended to the scanner buffer.

Every application must identify 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.

`NonTerminal` objects, symbolizing rule's left non-terminal symbols, represent each language rule. These objects are associated to a group of `Right` objects, which represent its rule's right sides. A `Right` object has two attributes:

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

Figure 5 shows a fragment of the object diagram representing the example shown above.

Any application code starts with its unique ID followed by its language name. The next code is an example of an application:

```
Application = "Very Simple App"
Language = "VerySimple"
       a=10;
       b=a;
       a;
       b;
```

Once the application's language specification has been translated into its respective object structure, a backtracking algorithm parsers the application's source code creating an abstract syntax tree (AST). Then, the initial non-terminal's

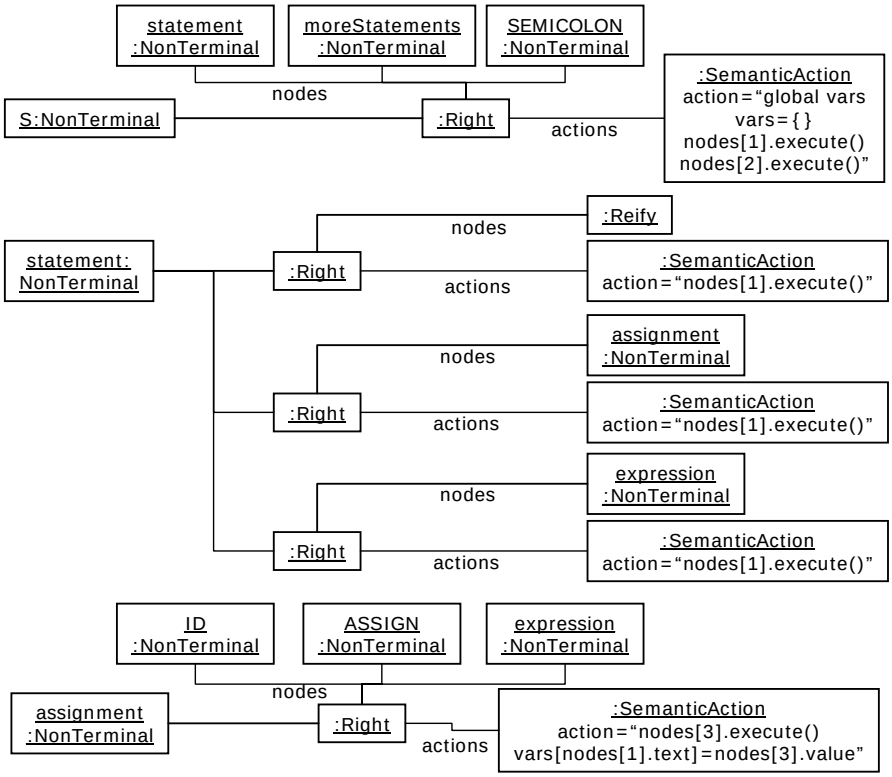

Fig. 5. Fragment of the language specification object diagram.

code is executed. The tree walking process is defined by the way grammar-symbols `execute` methods are invoked: the non-terminal `execute` method evaluates its associated semantic action. So, changes on language semantics may be automatically reflected on the applications being executed.

Interoperability between different applications –programmed in different languages– is achieved accessing the `nitrO` global object. Its attribute `applications` is a hash table of the existing applications in the system. Each `Application` object has two attributes:

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

6 Dynamic Application Adaptation

Accessing the `nitrO` object attributes, any application can adapt another one's behavior or structure at runtime, without any restriction and in a language-independent way. As a first example, we can use introspection to develop a trace routine that shows any running application symbol-table, regardless of its programming language:

```
Application = "Symbol Table"
Language = <#
Language=JustReflection
Scanner={}
Parser = {
"Initial Free-Context Rule"
  S -> _REIFY_ <#
nodes[1].execute()
#> ; }
Skip={ "\n"; "\t"; " "; }
NotSkip = {  }
#>

reify <#
# Shows any application symbol-table
def f(app,nitrO):
  if nitrO.apps.has_key(app):
    theApp=nitrO.apps[app]
    # Shows the Symbol Table in the
window
    nitrO.apps["Symbol
Table"].window.write(theApp.applicati
onGlobalContext)
  else:
    nitrO.shell.write(app+" must be
started.\n")
# Sets the function as a method
nitrO.apps["Symbol Table"].run=f

write("Routine installed as the run
method of Symbol Table application.")
#>
```

This application specifies itself its own programming language (lines 2 to 12): `JustReflection`, a unique `reify` statement (lines 13 to 25). The application file contains both the program source code and its language specification.

If we run this application, a program that is capable of showing any application-symbol table will be installed into the system –the message "*Routine installed as the run method of Symbol Table application*" will be shown. The `reify` statement defines a function (line 15) and afterwards sets it as an application method (line 23). This method takes an application ID as a parameter and searches the application object in

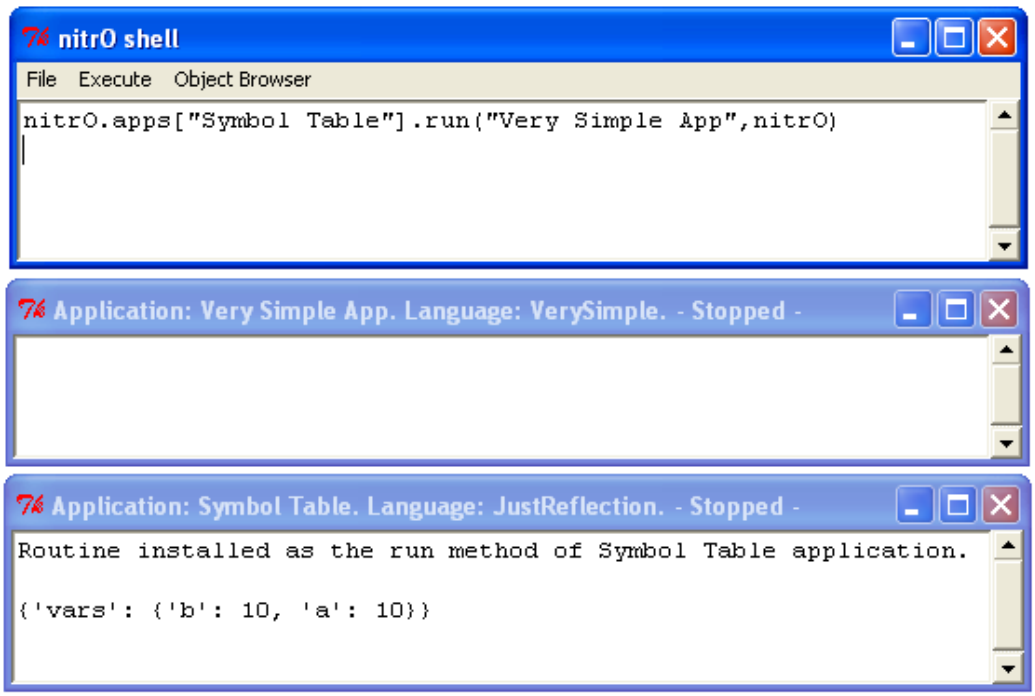

Fig. 6. Showing the runtime symbol-table of the "Very Simple App".

the system (lines 16 and 17). If it is found, application's symbol-table will be shown on its graphic window (line 19).

Any running application symbol-table could be displayed using the "Symbol Table" program, regardless of the language it has been written in. For instance, once the "Very Simple App" and the "Symbol Table" programs have been executed in the nitro system, we can show the "Very Simple App" application symbol table running the next statement in the nitrO shell:

```
nitrO.apps["Symbol Table"].run(
                "Very Simple App",nitrO)
```

The result is shown in figure 6. On the upper side we have the nitrO shell, where we can evaluate Python code accessing the `nitrO` object. Every running application has its own graphic window. The lower window shows the "Symbol Table" program execution; the one in the middle is the "Very Simple App" one. Evaluating the statement above, any application's symbol-table can be displayed whatever its programming language would be.

Following with the example presented in this paper, we will show how to achieve different levels of reflection in our system. The next group of `reify` sentences would dynamically adapt the running application, no matter which program or language might be used to execute them.

The next introspection example shows the existing variables of our simple program as well as their values:

```
reify <#
vars=nitrO.apps["Very Simple App"].
          ApplicationGlobalContext["vars"]
write( str(vars)+"\n" ) #Shows {b:10,a:10}
#>;
```

The following reify statement achieves structural reflection: takes the variables from the symbol table (line 2) and modifies (line 3), creates (line 4) and erases (line 5) symbol-table objects:

```
reify <#
        vars=nitrO.apps["Very Simple
        App"].applicationGlobalContext["vars"
        ]
        vars["a"]=vars["a"]*2 # Modifies "a"
        vars["c"]=0 # Creates a new variable
        del vars["b"] # Erases a variable
        #>;
```

We may enhance the assignment-statement semantics by showing a trace message every time an assignment takes place: computational reflection. Line 3 takes the assignment-statement syntactic rule. The code representing the new trace semantics is created in lines 4 to 6. Finally, line 8 enhances the assignment statement displaying a trace message:

```
reify <#
        from langSpec import SemanticAction
        assignment=nitrO.apps["Very Simple
        App"].language.syntacticSpec["assignm
        ent"]
        code="write(\"Assignment of
        +nodes[1].text"
        code=code+"with value
        +str(nodes[3].value)"
        code=code+".\\n\")"
        # Behavior adaptation
        assignment.options[0].actions.append(
        SemanticAction(code) )
        #>;
```

Using our platform, advanced adaptable systems written in real languages (e.g., Java and Python) are being developed. An example is an implicit-persistence system that makes runtime computational changes to the language semantics, doing transparent calls to persistence functions and making objects persist [13]. This system achieves great flexibility as no additional application code is needed to make it persist; changes can be dynamically made while the application is running and different levels of persistence can be selected for objects. At the same time, we are developing a dynamic-weaving aspect-oriented tool in which aspects can be set and unset to applications at runtime.

7 System Benefits

The non-restrictive reflective technique presented in this paper has the following advantages:

- The whole system is adaptable at runtime. Any system's feature can be adapted by means of the reflect statement, and there are no previously-defined restrictions imposed by any protocol.
- Expressiveness improvement. The way behavior is customized is not restricted to a framework that relies on method overriding – as happens with the use of MOPs. We offer a complete language (Python) that may be used to adapt any other language's feature.
- Language independence. The system can be programmed using any programming language. The inputs to our generic interpreter are both the application source code and the language specification.

- *What* can be reflected. Three levels of reflection are achieved at runtime: introspection, structural reflection and computational reflection.
- Application interoperability. Any application, whatever its programming language would be, may access, and reflectively modify, another program being executed. Therefore, there is no need to stop an application in order to adapt it at runtime: another application may be used to customize the former.

Our non-restrictive reflective technique can be used to develop or test at runtime any reflective or adaptable environment (e.g., fault-tolerant systems, adaptable operating systems, knowledge base systems or even web-based systems) without the necessity to modify the interpreter implementation. It might be also applied as a dynamic-weaving aspect-adaptation platform: as the back-end of an AOP tool that achieves dynamic inspection, selection and modification of reusable and language-independent crosscutting concerns [14].

7.1 Runtime Performance

The process of adapting an application at runtime, as well as the use of reflection, induces a certain overhead at the execution of a program.

Using interpreter optimization techniques such as just in time (JIT) compilation or adaptable native-code generation [15] has influenced on the extended commercial use of interpreted platforms (e.g., Java or Microsoft CLR).

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.

8 Conclusions

Most systems that offer computational reflection capabilities at runtime are based on the use of meta-object protocols (MOPs). MOPs give a system the ability to customize itself 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. Changing the MOP specification could involve different interpreter and language versions and, therefore, making the previous existing code been deprecated. Moreover, these systems use the same programming language at application and interpreter computational-environments, lacking cross-customization between different applications regardless of the programming language they have been written in. This paper describes a non-restrictive reflective technique capable of overcoming these limitations.

Using the structural reflection features of the Python programming language, we have developed a generic interpreter capable of interpreting every application written in any programming language. A language specification syntax has been defined in order to represent any context-free language.

The generic interpreter can obtain Python code (using the `reflect` statement) and evaluate it at the interpreter computational-environment: a real computational-environment jump is achieved, and no changes to the interpreter implementation have to be done. This mechanism may be used by an application to customize, at runtime, any program structure or behavior, without any restriction –no matter which programming language might be selected.

The final system is a computation platform that uses a non-restrictive reflective technique, can be programmed using any language, is completely adaptable at runtime, and has a great level of application interoperability. Therefore, it can be used to create or test highly adaptable environments based on dynamic separation of concerns.

The prototype source code, different language definitions and many testing applications can be freely downloaded from [12].

9 References

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J. M., and Irwin, J. 1997. Aspect Oriented Programming. *European Conference on Object-Oriented Programming Conference*, Finland, June 1997.

[2] IBM Research. Multi-Dimensional Separation of Concerns: An Overview". [Online]. Available: http://www.research.ibm.com

[3] Kiczales, G., Des Rivieres, J., and Bobrow, D. G. 1992. *The Art of Metaobject Protocol*. MIT Press.

[4] Ortin, F., and Cueva, J. M. Building a Completely Adaptable Reflective System. 2001.

ECOOP'2001. Workshop on Adaptive Object-Models and Metamodeling Techniques, Budapest, Hungary, June 2001.

[5] Chiba, S., and Michiaki, T. A Yet Another java.lang.Class. 1998. *ECOOP'98 Workshop on Reflective Object Oriented Programming and Systems*. Brussels, Belgium. July 1998.

[6] Kleinöder J., and Golm M. MetaJava: An Efficient Run-Time Meta Architecture for Java™. 1996. *International Workshop on Object Orientation in Operating Systems, IWOOOS'96*, Seattle, Washington, October 1996.

[7] Kiczales, G., Des Rivieres, J., and Bobrow, D. G. 1992. *The Art of Metaobject Protocol*. MIT Press.

[8] Douence, R., and Südholt, M. *The next Reflective 700 Object-Oriented Languages*. 1999. Technical Report 99-1-INFO, École des Mines de Nantes, Dept. Informatique, France.

[9] Rossum, G. *Python Reference Manual*. 2001. Fred L. Drake Jr. Editor. Relesase 2.1.

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

[11] Smith, B. C. *Reflection and Semantics a Procedural Language*. 1982. Ph. D. Thesis. Massachusetts Institute of Technology MIT/LCS/TR-272.

[12] Ortin, F. (2002, February) The Non-Restrictive Computational Reflective nitrO System. [Online]. Available: http://www.di.uniovi.es/reflection/lab/prototypes.html#nrrs

[13] Ortin, F., Martinez, A. B., Alvarez, D., and Cueva, J. M. A Reflective Persistence Middleware over an Object-Oriented Database Engine. 1999. *X!V Brazilian Symposium on Databases (SBBD)*, Florianopolis, Brazil, October 1999.

[14] Hürsch, W. L., and Videira Lopes, C. *Separation of Concerns*. 1995. Technical Report UN-CCS-95-03, Northeastern University, Boston, January 1995.

[15] Hölzle, U., and Ungar, D. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance 1994. *OOPSLA'94*, Portland, Oregon. October 1994.