The nitrO Reflective Platform

Francisco Ortín 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 of runtime reflective systems use meta-object protocols (MOPs). However, MOPs restrict the amount of features an application may 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 is nitrO, a non-restrictive reflective platform 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 the programming language they use.

Keywords*:* reflection, computational 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 reusable crosscutting aspects, creating the final application by *weaving* the program and its specific aspects. They 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 nitrO [4]: a non-restrictive reflective platform, in which 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 meta-object protocol reflective systems as well as its main drawbacks; 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 in section 7, and section 8 presents the final conclusions.

2 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 [5]. 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` inherited 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 would be called –showing a trace message and, therefore, customizing its message-passing semantics.

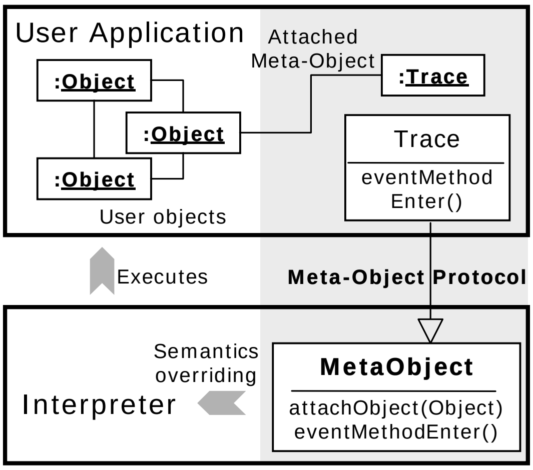


Fig. 1. MOP-based system architecture.

This Meta-Object Protocol reflective technique has different drawbacks:

1. The way a MOP is defined restricts the amount of features that may be customized [7]. 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, 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. One object's behavior may be overridden by attaching a meta-object to him. This meta-object may express the way it would modify its semantics by just overriding its super-class' methods – the interpreter will call this new method every time a message would be 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 to 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.

2.1 Python's Reflective Capabilities

We have selected the Python programming language [8] to develop our system because of its reflective capabilities [9]:

- 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 [10]: 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 the 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).

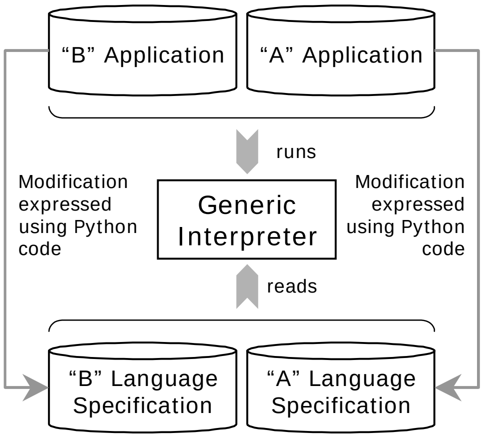

"B" Application
"A" Application
runs
Modification expressed using Python code
Generic Interpreter
Modification expressed using Python code
reads
"B" Language Specification
"A" Language Specification

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, 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 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.

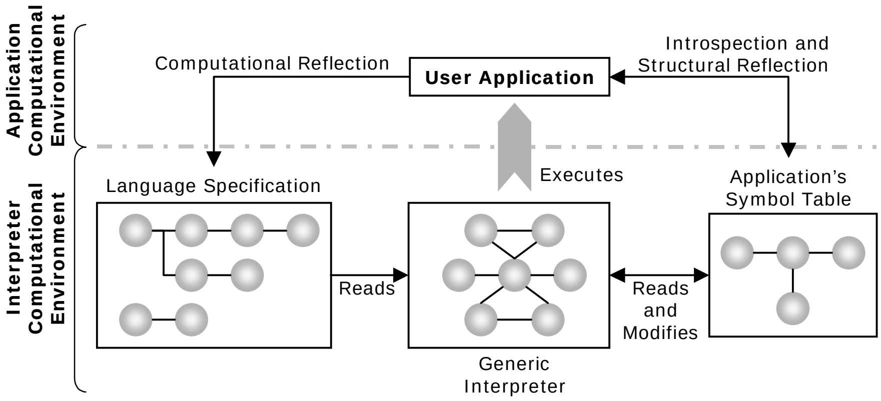

Application Computational Environment
Interpreter Computational Environment
Computational Reflection
User Application
Introspection and Structural Reflection
Executes
Language Specification
Application's Symbol Table
Reads
Reads and Modifies
Generic Interpreter

Fig. 3. Language specification and symbol table modification.

At runtime, any application may access one language specification by using the whole expressiveness of the Python programming language; there are no previous 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 uses the language specification while the application is running.

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.

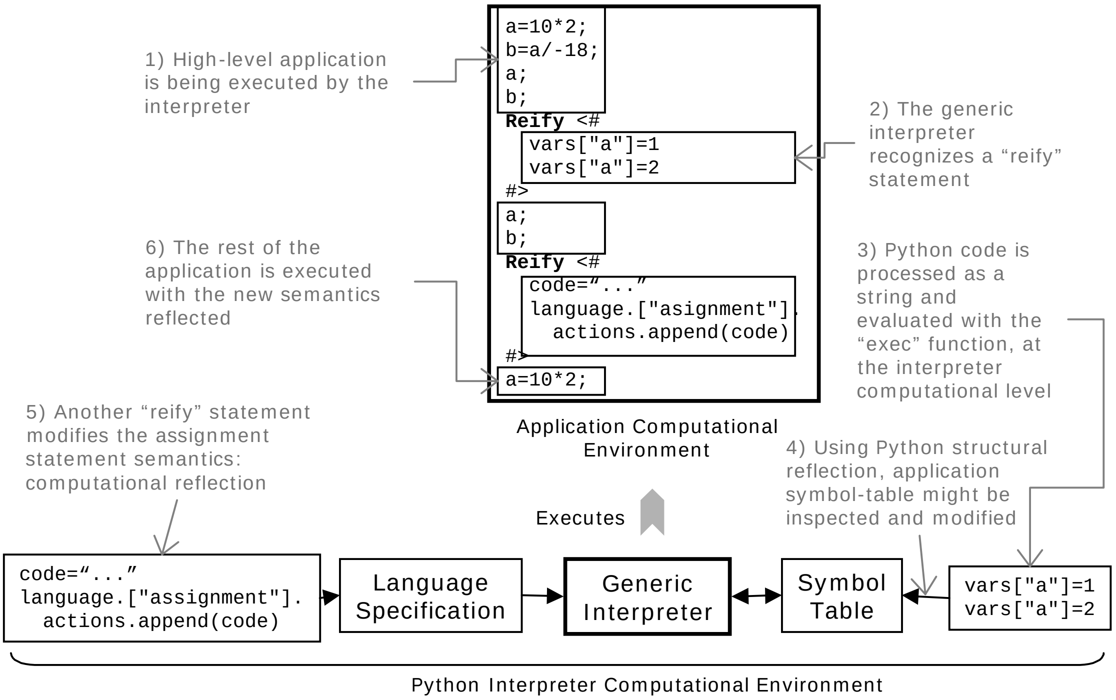

a=10*2;
b=a/-18;
a;
b;
Reify <#
vars["a"]=1
vars["a"]=2
#>
a;
b;
Reify <#
code="..."
language.["asignment"].
actions.append(code)
#>
a=10*2;
1) High-level application is being executed by the interpreter
2) The generic interpreter recognizes a "reify" statement
3) Python code is processed as a string and evaluated with the "exec" function, at the interpreter computational level
4) Using Python structural reflection, application symbol-table might be inspected and modified
5) Another "reify" statement modifies the assignment statement semantics: computational reflection
6) The rest of the application is executed with the new semantics reflected
Application Computational Environment
Executes
code="..."
language.["assignment"].
actions.append(code)
Language Specification
Generic Interpreter
Symbol Table
vars["a"]=1
vars["a"]=2
Python Interpreter Computational Environment

Fig. 4. Achieving a real computational jump.

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 to different language specifications and application's symbol tables.

Every language in our system includes the `reify` statement; the generic interpreter automatically recognizes it, no matter the language being used. Inside a `reify` statement Python code can 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 *what* language features could be adapted.

Looking for good performance, MOP-based systems simulate this 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.

5 Languages and Applications 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 and syntactic features are expressed by means of context-free grammar rules; their semantics, by means of Python code placed at the end of each rule. This is an example of a very simple language definition:

```
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
          |
          ;
  "Characacter 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 = {  }
```

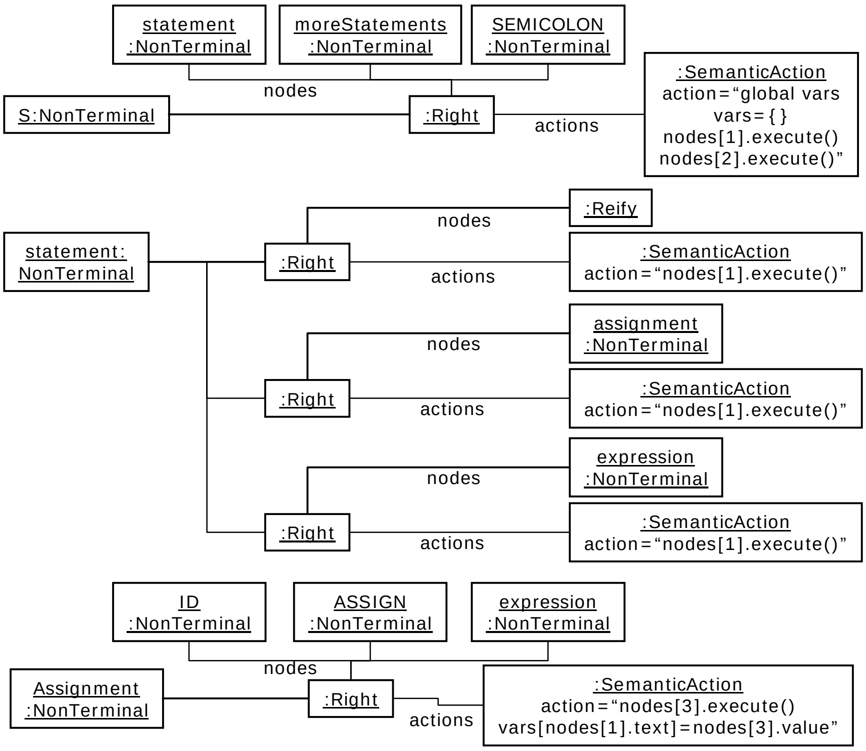
The `_REIFY_` reserved word indicates where a `reify` statement might be syntactically placed. Every application must identify its programming language previously to its source code. When the application is about to be executed, its respective language specification file is analyzed and translated into 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 that stores the Python code located at the end of each rule's description. 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. This is an example of an application:



statement :NonTerminal
moreStatements :NonTerminal
SEMICOLON :NonTerminal
nodes
S:NonTerminal
:Right
actions
:SemanticAction action="global vars vars={} nodes[1].execute() nodes[2].execute()"
statement: NonTerminal
nodes
:Reify
:Right
actions
:SemanticAction action="nodes[1].execute()"
nodes
assignment :NonTerminal
:Right
actions
:SemanticAction action="nodes[1].execute()"
nodes
expression :NonTerminal
:Right
actions
:SemanticAction action="nodes[1].execute()"
ID :NonTerminal
ASSIGN :NonTerminal
expression :NonTerminal
nodes
Assignment :NonTerminal
:Right
actions
:SemanticAction action="nodes[3].execute() vars[nodes[1].text]=nodes[3].value"

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

```
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 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 with 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. Following the example presented in this paper, the next group of `reify` sentences would dynamically adapt the running application, no matter which program or language might be used.

Our fist example shows the existing variables and its values: introspection.

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

Structural reflection means modifying, creating or erasing symbol-table objects:

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

We may enhance the assignment statement by showing a trace message every time an assignment takes place: computational reflection.

```
reify <#
from langSpec import SemanticAction
assignment=nitrO.apps["Very Simple App"].
      language.syntacticSpec["assignment"]
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) )
#>;
```

7 System Benefits

Our reflective system 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 previous 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 can be used to adapt any other language's feature.
- Language independence. The system may 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.

The result is a universal computation platform that may 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 application 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 [11].

Our higher adaptability technique has performance penalties. Future work will be focused on studying and implementing optimization techniques like just in time compilation to native code, combining interpreter and compiler techniques [12].

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 the programming language they have been coded in.

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 jump is achieved, and no changes to the interpreter implementation have to be done. This computational jump may be used by an application to customize, at runtime, any program structure or behavior, without any previous restriction –no matter which programming language might be selected.

The final system is a computation platform that can be programmed 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 based on dynamic separation of concerns.

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

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

[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] 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.

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

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

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

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

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