Adding Structural Reflection to the SSCLI

Francisco Ortin Jose M. Redondo Luis Vinuesa Juan M. Cueva
Computer Science Department, University of Oviedo
C/Calvo Sotelo s/n, 33007, Oviedo Spain
ortin@uniovi.es redondojose@uniovi.es vinuesa@uniovi.es cueva@uniovi.es

ABSTRACT

Although dynamic languages are becoming widely used due to the flexibility needs of specific software products, their major drawback is their runtime performance. Compiling the source program to an abstract machine's intermediate language is the current technique used to obtain the best performance results. This intermediate code is then executed by a virtual machine developed as an interpreter. Although JIT adaptive optimizing compilation is currently used to speed up Java and .net intermediate code execution, this practice has not been employed successfully in the implementation of dynamically adaptive platforms yet.

We present an approach to improve the runtime performance of a specific set of structural reflective primitives, extensively used in adaptive software development. Looking for a better performance, as well as interaction with other languages, we have employed the Microsoft Shared Source CLI platform, making use of its JIT compiler. The SSCLI computational model has been enhanced with semantics of the prototype-based object-oriented computational model. This model is much more suitable for reflective environments. The initial assessment of performance results reveals that augmenting the semantics of the SSCLI model, together with JIT generation of native code, produces better runtime performance than the existing implementations.

Keywords

Dynamic languages, structural reflection, prototype-based object-oriented computational model, Shared Source CLI, JIT code generation.

1. INTRODUCTION

Since the appearance of the first abstract machine (UNCOL, 1961 [Ste61]), the notion of using the specification of a computational processor without intending to implement it (abstract machine) has been used in many different contexts [Die00]. The main objective of the UNiversal Computer Oriented Language (UNCOL) was simplifying compilers development by employing a unique universal intermediate code.

A virtual machine involves a specific abstract machine implementation. The employment of specific abstract machines implemented by different virtual machines has brought many benefits to different computing systems. The most relevant are platform neutrality (USCD P-machine [Cla82] or Forth [Moo74]), application distribution (ANDF, Architecture Neutral Distribution Format [OSF91]), direct support of high-level paradigms (Smalltalk-80 [Gol83], SECD [Lan64] or Warren Abstract Machine [War83]) and application interoperability (PVM, Parallel Virtual Machine [Sun90]).

Despite of the many benefits obtained from using virtual machines, its main drawback has always been execution performance. Consequently, there has been considerable research aimed at improving the performance of virtual machine's application execution compared to its native counterparts. Techniques like adaptive Just In Time (JIT) compilation or efficient and complex garbage collection algorithms have reached such a point that Microsoft and Sun Microsystems identify this kind of platforms as appropriate to implement commercial applications. Nowadays, there are a lot of commercial languages and platforms that employ the concept of virtual machine to develop software products.

In parallel with the dominant virtual platforms (Sun's Java Virtual Machine and Microsoft .net) and its type-safe programming languages (Java and C#), a

.NET Technologies'2005 *conference proceedings*,
ISBN 80-86943-01-1

The development of is project is funded by Microsoft Research as the second Rotor Request for Proposals, awarded in 2004.

new approach of so called “dynamic languages” is emerging (examples are Python [Ros03], Ruby [Tho04] or Dylan [Sha96]). The main objective of these languages is to model the dynamicity that is commonly required in building software that is highly context-dependent due to the mobility of both the software itself and its users [ECO04]. Features such as meta-programming, reflection, mobility, dynamic reconfiguration and distribution are the domain of dynamic languages. Because of the benefits they offer, dynamic languages are employed in different scenarios such as adaptive programming [Mer03], dynamic aspect-oriented programming [Ort04] or high-level parallel software development [Hin03].

Dynamic languages, which also use abstract machine platforms, offer a much more relaxed type system at compile time that Java, C#, or any other type-safe language, in order to support their flexibility features –most part of the type system is dynamic. The unquestionable benefits of type-safe languages could still be obtained with unit testing suites that are currently widely used –as an example PyUnit is the Python version of the well-known JUnit testing framework. Using dynamic languages together with unit testing suites, the programmer can benefit both from the robustness of any type-safe language and from the flexibility of its dynamic features when needed [Mar03].

Dynamic Languages Performance

Looking for code mobility, portability and distribution facilities, dynamic languages usually employ the concept of abstract machine. Since their computational model offers dynamic modification of its structure and code generation at runtime, the existing virtual machine implementations are commonly developed by means of interpreters. Their flexibility and dynamicity capabilities make JIT native code generation (and its dynamic optimization) a complex task.

The existing implementations of Python for the Microsoft .net platform (Python for .Net from the Zope Community, IronPython, and the Python for .Net research project from ActiveState) have been developed as compilers that generate virtual machine’s intermediate code which simulates Python features over the .net platform. The implementations that have used the Java Virtual Machine (Jython or JPython) have also employed the same approach. Microsoft and Sun platforms were created to support static languages that do not offer structural reflective features such as adding attributes (fields) and methods at runtime. As these virtual machines do not support those primitives, additional code must be generated to support these features.

ActiveState tried to modify different free implementations of the .net platform in order to compile Python Programming Language to .net native code, but they abandoned the project because the abstract machine design “was not friendly to dynamic languages” [Ude03]. As Java and .net virtual machines have been designed taking into account their static features in order to obtain the highest runtime performance, it is difficult to add dynamic features to their existing implementations.

The main disadvantage of dynamic languages is 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 an application [Pop01]. However, as it happened with the implementation of Java Virtual Machine, speeding up the application execution of dynamic languages might facilitate their inclusion in commercial development environments. Since the research done by Hölzle and Ungar in dynamic JIT optimizing compilers applied to the Self programming language, virtual machine implementations have become faster by generating binary code at runtime [Höl94]. Nowadays, dynamic adaptive HotSpot optimizer compilers combine fast compilation and runtime optimizations of those parts of the code that are executed a higher number of times. These techniques have made virtual machines a real alternative to develop many types of software products.

The work presented in this paper employs these techniques to natively support dynamic languages over a virtual machine. We will show how we are incorporating reflective features to the Shared Source CLI implementation of the Microsoft .net platform. Adding dynamic reflective primitives to the platform internals will make it possible to compile dynamic languages directly to .net, obtaining performance benefits of using JIT native code generation. At the same time, applications developed in dynamic languages would be able to interoperate with any .net application or component, regardless of its programming language.

The rest of this paper is structured as follows. In the next section, we present the Microsoft Shared Source CLI and Section 4 introduces the set of reflective primitives to be added. Section 4 briefly describes the prototype-based object-oriented model as well as an analysis of how it can be incorporated to the SSCLI model. The specification of our new BCL reflective namespace is described in section 5 and section 6 summarizes the implementation issues. Finally, we analyze runtime performance (section 7) and section 8 presents the ending conclusions.

2. SHARED SOURCE CLI

The Microsoft SSCLI, Shared Source Common Language Infrastructure (also known as Rotor), is a source code distribution that includes fully functional implementations of both the ECMA-334 C# language standard and the ECMA-335 Common Language Infrastructure standard, various tools, and a set of libraries suitable for research purposes [Stu03]. The source code can be built and run under Windows XP, FreeBSD 4.5 or Mac OS X.

Rotor consists on 3.6 million lines of code that can be divided into 4 groups:

- The Execution Environment. This is the virtual machine of the .net platform that includes the JIT compiler, the garbage collector, the class loaders and the common type system.
- The Libraries. The SSCLI distribution includes the source code of its Base Class Library (BCL), runtime infrastructure and reflection (introspection) classes, networking and XML classes, and floating point and extended array libraries.
- Compilers and Tools. Rotor includes a C# compiler (ECMA-334) and a Jscript compiler written entirely in C#.
- Platform Abstraction Layer (PAL). This code implies the abstraction layer between the runtime environment and the operating system.

Excluding the PAL section, we have performed modifications and enhancements in every part of the Rotor structure to develop our project.

3. EXTENDING THE REFLECTIVE CAPABILITIES OF ROTOR

Reflection has been recognized as a suitable tool to aid the dynamic evolution of running systems, being the primary technique to obtain the metaprogramming, adaptiveness, and dynamic reconfiguration features of dynamic languages [Caz04]. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [Mae87]. In a reflective system, its computational domain is enhanced by its own representation, offering at runtime its structure and semantics as computable data.

The main criterion to categorize runtime reflective systems is taking into account what can be reflected. Following this classification, we can distinguish:

- Introspection: The system's structure can be consulted but not modified. Both Java and .net platforms offer this level of reflection. By means of the `java.lang.reflect` package (Java) and `System.Reflection` namespace (.net), the programmer can get information about classes, objects, methods and fields at runtime.
- Structural Reflection: The system's structure can be modified and the changes should be reflected at runtime. An example of this kind of reflection is the Python feature of adding fields –attributes– or methods to both objects and classes.
- Computational (Behavioral) Reflection: The system semantics can be modified, changing the runtime behavior of the system. For instance, metaXa –formerly called MetaJava [Gol97]– is a Java extension that offers the programmer the ability to dynamically modify the method dispatching mechanism. The mechanism most commonly employed in this level of reflection is Meta-Object Protocols (MOPs) [Kic91].

As mentioned above, the runtime reflective features of Rotor are restricted to the introspection level. However, the .net platform offers the facility to dynamically generate CIL code at runtime in a limited way (it only permits to create new types, not adding new methods to the existing classes) by means of its `System.Reflection.Emit` namespace.

Dynamic languages offer the structural level of reflection in their computational model. This level of reflection is the one employed by dynamic languages to develop adaptive software. Much research on MOPs has revealed that computational reflection suppose a huge performance penalty in comparison with the benefits it provides [Tan03]. At the same time, many behavioral features could be simulated with structural reflection (e.g., adapting method invocation semantics could be substituted by a method wrapping service developed with structural reflection).

Reflective Facilities

We have extended the .net CLI with a set of structural reflective primitives extensively used in dynamic languages, at the abstract machine level. A new namespace has been added to the Base Class Library (BCL): `System.Reflection.Structural`. We will show in Section 5 which are its specific primitives, but its functionality can be grouped into:

- Attributes manipulation. It can be modified not only the structure of a class (altering the structure of all of its instances), but the composition of a single object. Attributes may be added, deleted or replaced.
- Methods manipulation. Methods of classes can be added and erased dynamically. Therefore, the set of messages accepted by an object could change at runtime depending on their dynamic

context. At the same time, a new method could be placed in a sole object. The body of these new methods can be obtained as copies of the existing ones, or it dynamically generated by means of the `System.Reflection.Emit` namespace.

The programmer could combine these facilities with the introspective services already offered by the .net platform, making the CLI an ideal system to develop language neutral adaptive software.

Conceptual Problems

There exist conceptual inconsistencies between the class-based object-oriented computational model and structural reflective facilities. These inconsistencies were detected and partially solved in the field of object-oriented database management systems. In this area, objects are stored but their structure or even their types (classes) could be altered afterwards, as a result of software evolution [Ska87].

The first scenario of modifying class's structure (attributes) implies updating the composition of every object that is an instance of this class. This mechanism was defined as schema evolution in the database field. The modification of class's instances could be performed when the class is modified (eager) or when the object is up to be used (lazy) [Tan89]; it is only necessary to know at runtime the class an object is instance of. The dynamic evolution of class's methods and attributes can produce situations where code access to attributes or methods that do not exist in a specific execution point; these situations should be checked by a dynamic type checking mechanism, employing exception handling.

Another possibility that a reflective model supports is much more difficult to be modeled in a class-based language. How can an object's structure be modified without altering the rest of its class's instances? This problem was detected in the development of MetaXa, a reflective Java platform implementation [Gol97]. The approach they chose was the same as the adopted by some object-oriented database management systems: schema versioning [Rod95]. A new version of the class (called "shadow" class in MetaXa) is created when one of its instances is reflectively modified. This new class is the type of the recently customized object.

This model causes different problems such as maintaining the class data consistency, class identity, using class objects in the code, garbage collection, inheritance or memory consumption, involving a really complex implementation difficult to manage [Gol97]. One of the conclusions of their research was that the class-based object-oriented model does not fit well to structural reflective environments. They finally stated that the prototype-based model would express reflective features better than class-based ones [Gol97].

4. PROTYPE-BASED OO MODEL

In the prototype-based object-oriented computational model the main abstraction is the object, suppressing the existence of classes [Bor86]. Although this computational model is simpler than the one based on classes, there is no loss of expressiveness; i.e. any class-based program can be translated into the prototype-based model [Ung91]. A common translation from the class-based object-oriented model is by following the next scheme (Figure 1):

- Similar object's behavior (methods of each class) can be represented by trait objects. Their only members are methods. Thus, their derived objects share the behavior they define.
- Similar object's structure (attributes of each class) can be represented by prototype objects. This object has a set of initialized attributes that represent a common structure.
- Copying prototype objects (constructor invocation) is the same as creating instances of a class. A new object with a specific structure and behavior is created.

In class-based languages where classes are first class

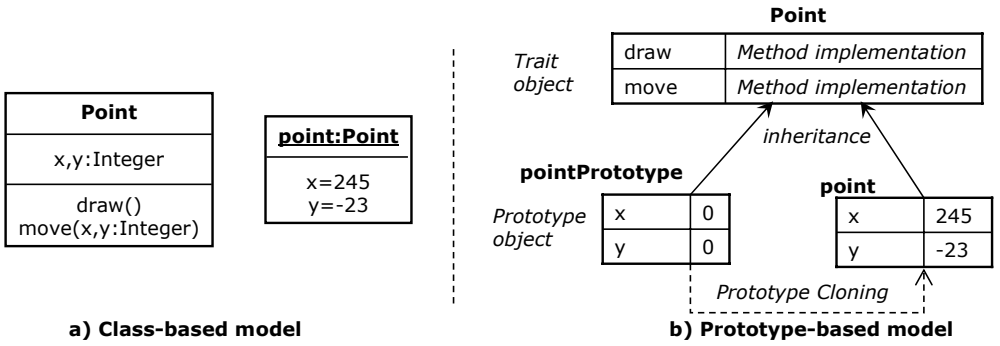

Figure 1. Translation between the class and prototype based computational model.

objects (Java, Smalltalk or C#), classes are represented by objects at runtime (e.g., in the .net platform instances of `System.Type` are objects that represent classes or another type). This demonstrates that, besides not existing loss of expressiveness, the translation of the model is intuitive and facilitates application interoperability, no matter whether the programming language uses classes or not. This is the reason why this model has been previously considered as a universal substrate for object-oriented languages [Wol96].

Finally, this object-oriented computational model does model structural reflective primitives in a consistent way –structural reflective languages such as Moostrap [Mul93] or Self [Ung87] have employed this model in a successful way [Ort05]. The prototype-based object model overcomes the schema versioning problem stated in Section 3.2. Modifying the structure (attributes as well as methods) of a single object is performed directly, because any object maintains its own structure and even its specialized behavior. As shared behavior is placed in trait objects, its customization implies the adaptation of types (schema versioning).

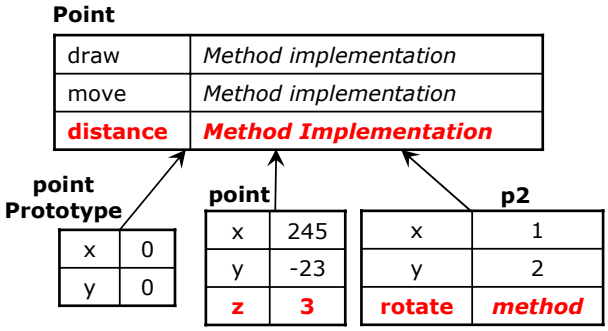

Figure 2. Structural reflective modification of objects.

Figure 2 shows an example scenario. The initial `point` and `p2` objects are clones of the `pointPrototype` and their shared behavior is placed in the `Point` trait object. A new coordinate attribute has been added only to the `point` object. Employing the same approach, only the `p2` object is capable to `rotate` its coordinates. Finally, all the derived objects from the `Point` trait object will be able to use the new `distance` method.

Adapting Rotor's Computational Model

We have seen how the prototype-based object-oriented model is capable of modeling structural reflection in a coherent way. However, the .net platform employs a class-based model all over the CLI. Moreover, if we want to interoperate with any existing .net language or application, we must follow the class-based model. Therefore, our approach consists on continue using classes but the reflective primitives will offer a semantic of a prototype-based object model:

- As classes are first class objects in the .net platform, their structure is customized by means their `System.Type` instances. Altering their methods produces adaptation of shared behavior as if we were modifying a trait object in the prototype-based object model. In case we adapt attributes of `System.Type` objects, what we obtain is the customization of all the existing instances of the class adapted (schema evolution). Looking for a good runtime performance, we have developed a lazy schema evolution mechanism [Tan89].
- Objects are treated as prototypes. Although in the class-based object model it is not possible to add specific behavior to a single object, neither to modify its attributes without adjusting its class structure, we permit to apply these structural reflective services to a specific class instance. Employing this model, we can dynamically add or erase both methods and attributes to a specific object, overcoming the abovementioned schema versioning problem. Of course, any compiler of a statically type-checked .net language (e.g., C#) needs to be modified to make the most of these reflective features; dynamic languages will employ them directly.

As an example, we show in Figure 3 a Python syntactic approach of a program that uses this combination of the class-based and prototype-based object model, when employing the structural reflective primitives (last feature shown in the example code is not really supported by the Python programming language).

We first create a `Point` class with its constructor and the `move` and `draw` methods. An instance is then created (`point`) and a `draw` message passed. Then we modify the structure of a single object adding a new `z` attribute and its respective `draw3D` method. Finally, we add a new behavior to the `Point` class (the `getX` method) and a new `isShowing` field to every `Point` instance, obtaining the schema evolution mechanism previously described.

5. EXTENDING THE BCL

The structural reflective features mentioned above require the enhancement of the .net platform semantics. We have first implemented all of them in a new namespace called `System.Reflection.Structural`. The primitives were initially developed in C#, making extensive use of the .net's introspection facilities. This first implementation has empirically demon-

strated the viability of the proposed computational model, giving us a first assessment of performance.

```
class Point:
  "Constructor"
  def __init__(self, x, y):
    self.x=x
    self.y=y

  "Move Method"
  def move(self, relx, rely):
    self.x=self.x+relx
    self.y=self.y+rely

  "Draw Method"
  def draw(self):
    print "("+str(self.x)+
          ","+str(self.y)+")"

point=Point(1,2)
point.draw()     # (1,2)

# Modify attributes of a single object
point.z=3
print point.z       # 3

# Modify methods of a single object
def draw3D(self):
  print "("+str(self.x)+
        ","+str(self.y)+
        ","+str(self.z)+")"

point.draw3D=draw3D
point.draw3D()       # (1,2,3)

# Modify methods of a class
def getX(self):
  return self.x

Point.getX=getX
print point.getX()   # 1

# Modify attributes of
# every Point instance
Point.isShowing=0
```

Figure 3. Example Python code using structural reflection services.

This is a resume of the most significant elements we have added to the BCL (all of them, static methods of the `Structural` utility class):

- `addMethod` and `removeMethod` methods receive an object or class (`System.Type`) as a first parameter indicating whether we want to modify a single object or a shared behavior. The second parameter is a `MethodInfo` object of the `System.Reflection` namespace. This object uniquely describes the identifier, parameters, return type, attributes and modifiers of a method. The user could create a new method employing the `System.Reflection.Emit` namespace, and add it to an object or class by means of its `MethodInfo`.
- The `invoke` primitive executes the method of an object or class specifying its name, return type and parameters. When the programmer uses the `invoke` method to pass a message to an object, it is checked if the object has a suitable method. In case it exists, it is executed; otherwise the message is passed to its class (its trait object). A `MissingMethodException` is thrown if the message has not been implemented.
- The `addField`, `getField` and `removeField` methods are used to modify the runtime structure of single objects or their common schema (classes). If the first parameter is an object, the rest of instances of its class will not be modified. Adding a field to a class ensures that all of the existing instances contain the specified attribute; removing it guarantees that none have that attribute.

Employing these primitives, the code in Figure 4 shows the C# version of the Python reflective program of Figure 3.

```
RuntimeStructuralFieldInfo rsfi = new Run-
    timeStructuralFieldInfo("z",
    typeof(int),3, FieldAttributes.Public);
Structural.addField(point,rsfi);
// Draw3D is the MethodInfo a new method
// created with System.Reflection.Emit
Structural.addMethod(point,draw3D);
Object[] pars={};
Structural.invoke(point,draw3D.ReturnType,
    "draw3D",pars);
// getX is another MethodInfo object
Structural.addMethod(typeof(Point),getX);
Console.WriteLine(Structural.invoke(
    point,getX.ReturnType,"getX",params) );
rsfi = new RuntimeStructuralFieldInfo(
    "isShowing", typeof(bool),false, FieldAt-
    tributes.Public);
Structural.addField(typeof(Punto), rsfi);
```

Figure 4. C# structural reflective program.

We have implemented other useful primitives such as {`field`, `method`}`Exists`, `getFieldValue`, `alter`{`Method`, `Field`} or `getMethod`, as well as additional classes such as `RuntimeStructucturalFieldInfo` or {`Member`, `Method`, `Field`}`Collection`. Now that we have confirmed that this set of primitives are suitable to offer the adaptable computational model presented, we are implementing part of them as an enhancement of the semantics of specific CIL instructions. As an example, the `invoke` primitive should not be only part of the BCL interface; its semantics must also be included in the `call` and `callvirt` CIL statements. In order to achieve this functionality we are also modifying the semantic analysis module of the SSCLI C# compiler –it should be allowed to invoke non-existing methods at compile time.

6. IMPLEMENTATION

The complexity of Rotor implementation prevented us from directly implementing the operations inside the runtime environment. A set of steps were defined to gradually incorporate structural reflection in Ro-

tor. Modifying different parts of the system one by one –from BCL to the binary code generated at runtime– has allowed us to refine the model using the experience gained.

We have divided the development process into three steps:

- Step 1: BCL-level implementation. In this step we have implemented all the reflective primitives in C#, making use of .net introspective capabilities. The runtime environment was not modified in this step. The programmer should use all the reflective features explicitly by means of the BCL.
- Step 2: VM-level implementation. In this second step we have moved the implementation of the BCL primitives developed in the previous step to an equivalent C implementation, included into the execution environment. The BCL interface was not modified, but the implementation was included inside the virtual machine getting significantly better runtime performance. We used Rotor internal structures, data types and routines to our advantage.
- Step 3: JIT-level implementation. The final step in our development has been modifying the Rotor JIT code generation mechanism. We have extended some CIL instruction semantics modifying the code generated by the JIT, in order to support structural reflection of existing .net applications.

The Step 1 implementation employs a central data structure that uses four C# hash-tables to store relationships between added members and their owners (either classes or instances). When accessing members, these hash-tables are consulted first and, if the member has not been reflectively added, the rest of the process continues searching in the class hierarchy using introspection. If the top of the hierarchy is reached without finding the appropriate member, a `MissingMemberException` is thrown. This implementation is much easier than developing this code inside the runtime environment, but its execution performance is significantly slower.

Once the first step was developed and tested, we proceeded to include the implementation of these reflective services inside the execution environment. The most important decision to be done was finding the suitable place to put the data structure that stored the reflective information. Since direct object structure manipulation turns to be much more difficult than we expected, due to its fixed-size object design, we decided to use each object's `Syncblock` to store reflective data. Thus, we assigned each object (and class) a specific structure to store its reflective information.

Although the VM-level implementation improved runtime performance considerably, reflective behavior must still be explicitly stated by the programmer. That is to say, it is not possible to reflectively adapt legacy .net binary code, because structural reflection must be explicitly used. We are currently working on overcoming this lack, implementing the third step of the development process.

Project Status

Structural reflective primitives are being included into the CIL instruction semantics (3rd step). We have already included the attribute-manipulation ones. The new semantics has already been added to the `ldfld` and `stfld` CIL statements of the platform.

The main idea to achieve the new behavior is to modify the native code generated by the JIT compiler. Instead of the original code that uses statically calculated member offsets, we generate a call to a helper function. This function explores the object structure in order to calculate member addresses using the reflective data included in the object's `Syncblock`.

Finally, we are working on modifying the JIT compiler to support reflective manipulation of methods. Our planned implementation will generate code to a new helper function, which will return the method address (depending on the stored reflective information), performing the invocation of the returned address.

7. PRIMARY PERFORMANCE ASSESSMENT

The use of a JIT compiler in a reflectively adaptive environment could introduce performance benefits in comparison with existing interpreted-based implementations. We have measured the performance of our second step implementation in comparison with four well-know Python platforms. This assessment will give us an idea of the benefits that could be obtained in the future.

We have measured execution time of all the primitives described above in loops of 10,000 iterations, removing any I/O and graphical routines [Bul00]. All tests were carried out on a lightly loaded 3.2 GHz iPIV hyper-threading system with 1 Gb of RAM running WindowsXP.

The specific well-known Python implementations used in our tests were:

- CPython 2.4 (commonly referred as simply Python): This is probably the most widely used Python interpreted implementation; it is called CPython because it has been developed in C.

Primitive	Jython	IronPython	BCL	ActivePython	CPython
1. Add `int` attributes to an object	20,679	36,032	1,632	590	541
2. Add `Object` attributes to an object	20,290	32,013	1,762	611	580
3. Add `int` attributes to a class	20,063		440	551	591
4. Add `Object` attributes to a class	20,320		460	661	610
5. Delete `int` attributes from an object	18,406		971	561	591
6. Delete `Object` attributes from an object	19,028		961	611	601
7. Delete `int` attributes from a class	18,536		200	540	561
8. Delete `Object` attributes from a class	18,896		210	581	560
9. Access attributes from an object	18,607	23,000	530	521	530
10. Access non-existing attributes from an object	20,019	21,017	1,191	641	601
11. Access attributes from a class	18,577		150	511	481
12. Access non-existing attributes from a class	20,028		370	611	571
13. Add methods to an object	22,592	30,230	3,364	640	480
14. Add methods to a class	23,192		2,000	720	560
15. Invoke methods added to an object	20,624	24,010	3,564	760	600
16. Invoke non-existing methods to an object	25,276	25,567	1,840	720	804
17. Invoke methods added to a class	21,064		2,680	720	680
18. Delete methods added to an object	18,504		1,240	520	520
19. Delete methods added to a class	18,464		280	520	520

Table 1. Measurement of 10,000 calls to each reflective primitives.

- ActivePython 2.4.0. Another interpreted distribution of Python (from ActiveState) available for Linux, Solaris and Windows.
- Jython 2.1 (formerly called JPython): A 100% pure Java implementation of the Python programming language. It is seamlessly integrated with the Java 2 Platform.
- IronPython 0.6: is a new promising implementation of the Python language targeting the Common Language Runtime (CLR). It compiles python programs into CIL bytecodes that run on either Microsoft's .net or the Open Source Mono platform. Its current release is a pre-alpha 0.6 version.
  We have not used Zope's Python for .net because it is not really the same approach as Jython in Java; it provides an implementation of the Python language and runtime engine in pure Java. Python for .net is not a re-implementation of Python, just an integration of the existing CPython runtime with .NET. Neither have we employed ActiveState Python for .net because they have quit this research project caused by the poor performance results obtained [Ude03].

Table 1 shows the measurement of each primitive execution called 10,000 times, expressed in milliseconds. As we can appreciate in this table, Jython and IronPython obtain the worst performance results in all of the tests –IronPython do not implement deletion of members, neither class manipulation. The requirement to implement Jython as a 100% pure Java offers a great interoperability with any Java program, but it causes a significant performance penalty. The same happens to IronPython: generating CIL code that simulates the Python reflective model over a platform that does not support it produces low performance at runtime. Probably, this performance penalty is caused by the amount of extra code that must be generated to support the reflective model.

Figure 5 and Table 1 show how our BCL implementation of structural reflective primitives is faster than the two systems that generate intermediate code: Jython and IronPython. Note than, since the range of values of Jython and IronPython are considerably different from the rest of implementations, we have separated both scales in Figure 5. Therefore, the values of these two implementations are shown on the right of the figure, whereas the rest appear on the left. Our BCL implementation is more than 30 times faster than Jython.

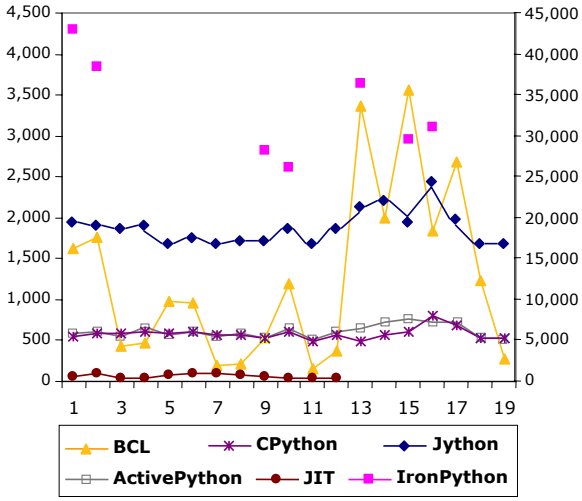

Figure 5. Execution time (milliseconds) of each primitive over different implementations.

Figure 5 also illustrates how our system performance is not as good as the native interpreter implementation (CPython and ActiveState). However, the BCL implementation is the fastest when modifying class's structure. This is due to the laziness of the schema-evolution mechanism we have implemented.

Best results are obtained by the two platforms that interpret the Python code by means of a C implementation: ActiveState and CPython. Both obtain quite similar results, which are significantly better than the BCL version when using objects –the most typical scenario– but worse when employing classes.

As we have mentioned above, we are currently including the structural reflective primitives into the JIT compiler. Although the project is still in an immature point to release definitive performance results, we have enough information to get a first interesting estimation. Executing the same test suite with the new attribute semantics added to the SSCLI runtime environment, employs the 15.76% average time in comparison with the BCL version (the new implementation is 10.88 times better that the first one). Furthermore, the execution of JITted structural reflective primitives requires an average of 11.58 % time in comparison with the time required in CPython. Figure 5 shows these values graphically (JIT).

Although we have not developed the addition and deletion of methods in objects and classes, these first results give us an initial estimation of how the use of a JIT compiler can be employed to obtain good performance of runtime adaptive applications. Certainly, since we have only developed part of the reflective computational model of Python –e.g. we have not implemented the Python feature of modifying the class an object is instance of–, the results obtained could not be directly compared with execution performance of complete implementations of the Python programming language. What our work has revealed is that JIT compilation techniques are really appropriate to improve the performance of adaptive systems and languages. The key point is to modify the semantics of the abstract machine instead of generating intermediate code that simulates this adaptive behavior. Adding this semantics at the JIT compiler level is complex task, but appears to be worth the effort.

8. CONCLUSIONS

Abstract machines have been widely employed to design programming languages because of the many advantages they offer. Although performance was the main drawback in the past, modern techniques like adaptive (hotspot) Just In Time compilation have overcome this weakness. That is one of the reasons why virtual machine platforms are nowadays commercially used.

Currently, due to the special flexibility and adaptively needs of specific systems, the so called "dynamic languages" are becoming more and more used. These languages also make use of the process of compilation to an abstract-machine's intermediate code. However, due to the complexity of its flexible semantics, the virtual machine is commonly developed as an interpreter.

Looking for better performance results, there have been different attempts to implement Python and other highly dynamic languages for.net and Java platforms. They have resulted in systems with really poor performance, so bad that they were considered unusable. Some of them have abandoned this idea. We have evaluated two, Jython and IronPython, in comparison with other two interpreted versions – CPython and ActivePython. The interpreted versions were much faster than the JIT compiled ones, when measuring their reflective features. Despite these results, we think that the use of a JIT compiler in reflective adaptive environments could obtain better performance than interpreting the intermediate language. Since Java and .net platforms have not been designed with that purpose, modifying the semantics of the abstract machine (and, therefore, the implementation of the virtual machine) might produce the expected benefits.

In order to obtain a first performance assessment, we have developed a set of structural reflective primitives as part of the BCL .net platform. These primitives implement the semantics of the prototype-based object-oriented model over the SSCLI class-based platform. This first implementation obtains better performance results that generating CIL code, because implies quite less code to execute at runtime.

Finally, we have performed an initial assessment of performance results obtained with the inclusion of part of the structural reflective primitives into the SSCLI runtime environment. This initial evaluation gives us an estimation of the performance benefits that will be obtained in the future, when the whole reflective semantics will be included in the code generated by the JIT compiler. Although we have only added part of the reflective features of the Python programming language, the assessment presented reveals that using an adaptive-designed platform in conjunction with a JIT compiler involves important performance benefits to implement dynamic languages.

9. REFERENCES

[Bor86] Borning, A.H. Classes versus Prototypes in Object-Oriented Languages. In Proceedings of

the ACM/IEEE Fall Joint Computer Conference, pp. 36-40, 1986.

[Bul00] Bull, J. M., Smith, L.A., Westhead, M.D., Henty, D.S., and Davey, R.A. A Benchmark Suite for High Performance Java. Concurrency: Practice and Experience 12, pp. 375-388, 2000.

[Caz04] Cazzola, W., Chiba, S., and Saake, G. In ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution. Research Reports on Mathematical and Computing Sciences. Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2004.

[Cla82] Clark, R., and Koehler, S. The UCSD Pascal Handbook. Prentice-Hall, Englewood Cliffs, 1982.

[Die00] Diehl, S., Hartel, P., and Sestoft, P. Abstract Machines for Programming Language Implementation. Elsevier Future Generation Computer Systems, Vol. 16(7), 2000.

[ECO04] ECOOP'04 Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity. ECOOP 2004 Workshop Reader, Lecture Notes in Computer Science, Vol. 3344, Oslo, Norway, 2004.

[Gol83] Goldberg, A., and Robson, D. Smalltalk-80, The Language and Its Implementation. Addison-Wesley, Reading, MA, 1983.

[Gol97] Golm, M., and Kleinöder, J. MetaJava - A Platform for Adaptable Operating- System Mechanisms. Lecture Notes in Computer Science 1357, Springer-Verlag, pp.507-507, 1997.

[Hin03] Hinsen, K. High-Level Parallel Software Development with Python and BSP. Parallel Processing Letters, Vol. 13, No. 3, pp. 473-484, 2003.

[Höl94] Hölzle, U., and Ungar, D. A Third-Generation Self Implementation: Reconciling Responsiveness with Performance. In Proceedings of the ACM OOPSLA Conference, Portland, OR, 1994.

[Kic91] Kiczales, G. The Art of the Metaobject Protocol. The MIT Press, 1991.

[Lan64] Landin, P.J. The mechanical evaluation of expressions. Computer Journal 6 (4), pp. 308-320, 1964.

[Mae87] Maes, P. Computational Reflection. PhD. Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium, 1987.

[Mar03] Martin, R.C. Are Dynamic Languages Going to Replace Static Languages? Artima Developer, April, 2003.

[Mer03] Mertz, D., and Simionato, M. Metaclass Programming in Python – Pushing Object-Oriented Programming to the next level. IBM depeloperWorks. February 2003.

[Mor74] Moore, C. Forth: A new way to program a mini-computer, Astronomy & Astrophysics Supplement: 15, pp. 497-511, 1974.

[Mul93] Mulet, P., and Cointe, P. Definition of a Reflective Kernel for a Prototype-Based Language. In the International Symposium on Object Technologies for Advanced Software, Kanazawa (Japan), 1993.

[Ort04] Ortin, F., and Cueva, J.M. Dynamic Adaptation of Application Aspects. Journal of Systems and Software 71(3), pp. 229-243,2004.

[Ort05] Ortin, F., and Diez, D. Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection. Information and Software Technologies 47(2), pp. 81-94, 2005.

[OSF91] Open Systems Foundation. OSF Architecture-Neutral Distribution Format Rationale, 1991.

[Pop01] Popovici, A., Gross, Th., and Alonso, G. Dynamic Homogenous AOP with PROSE, Technical Report, Department of Computer Science, ETH Zurich, Switzerland, 2001.

[Rod95] Roddick, J. A Survey of Schema Versioning Issues for Database Systems. Information and Software Technology 37 (7), 383-393, 1995.

[Ros03] Rossum, G.V., Fred, L., and Drake, Jr. The Python Language Reference Manual. Network Theory, 2003.

[Sha96] Shalit, A., Moon, D., and Starbuck, O. The Dylan Reference Manual. Addison-Wesley, 1996.

[Ska87] Skarra, A.H., and Zdonik, S.B. Type Evolution in an Object-Oriented Database. Research Directions in Object-Oriented Programming, MIT-press, pp. 393-415, 1987.

[Ste61] T.B. Steel Jr. A first version of UNCOL. In Western Joint Computing Conference, pp. 371–378. New York, 1961.

[Stu03] Stutz, D., Neward, T., and Shilling, G. Shared Source CLI Essentials. O'Reilly, 2003.

[Sun90] Sunderam, V. S. PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice and Experience 2(4), pp 315-339, 1990.

[Tan89] Tan, L., and Katayama, T. Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution. In Proceedings of First International Conference on Deductive and Object-Oriented Databases, DOOD, pp. 241-258. 1989.

[Tan03] Tanter, E., Noyé, J., Caromel, D., and Cointe, P. Partial behavioral reflection: spatial and temporal selection of reification. In Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). California, USA, 2003.

[Tho04] Thomas, D., Fowler, C, and Hunt, A. Programming Ruby. 2nd Edition. Addison-Wesley Professional, 2004.

[Ude03] Udell, J. Dynamic languages and virtual machines. InfoWorld, August, 2003.

[Ung87] Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In OOPSLA Conference Proceedings. SIGPLAN Notices, 22 (12), pp. 227-241, 1987.

[Ung91] Ungar, D., Chambers, D., Chang, B.W., and U. Hölzl. Organizing Programs without Classes. Lisp and Symbolic Computation, Kluwer Academic Publishers, pp. 223-242, 1991.

[War83] Warren, D.H.D. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, 1983.

[Wol96] Wolczko, M., Agesen, O., and Ungar, D. Towards a Universal Implementation Substrate for Object-Oriented Languages, Sun Microsystems Laboratories, 1996.