Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection [⋆]

Francisco Ortin [*], Diego Diez

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

Abstract

The concepts of abstract and virtual machines have been used for many different purposes to obtain diverse benefits such as code portability, compiler simplification, interoperability, distribution and direct support of specific paradigms. Despite of these benefits, the main drawback of virtual machines 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 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.

What we have noticed in our research work is that these platforms have heterogeneity, extensibility, platform porting and adaptability limitations caused by their monolithic designs. Most designs of common abstract machines are focused on supporting a fixed programming language and the computation model they offer is set to the one employed by the specific language. We have identified reflection as a basis for designing an abstract machine, capable of overcoming the previously mentioned limitations. Reflection is a mechanism that gives our platform the capability to adapt the abstract machine to different computation models and heterogeneous computing environments, not needing to modify its implementation. In this paper we present the reflective design of our abstract machine, example code extending the platform, a reference implementation, and a comparison between our implementation and other well-known platforms.

Key words: Reflection, Abstract Machine, Virtual Machine, Adaptability, Heterogeneity

⋆ Partially funded by the *Asturian Foundation of Applied Scientific Rersearch.*

* Tel.: +34-985-10-3172. FAX: +34-985-10-3382.
Email address: `ortin@lsi.uniovi.es` (Francisco Ortin).
URL: `http://www.di.uniovi.es/~ortin` (Francisco Ortin).

1 Introduction

An *abstract machine* is the specification of a computational processor without intending to implement it in hardware. Common to most abstract machines are a program store and state, usually including a stack and registers, and they are defined by means of its programming-language's operational semantics. The term *virtual machine* is commonly used to denote a specific abstract machine implementation (sometimes called *emulator, interpreter* or *runtime*), in similar way as we use the term program for implementations of an algorithm.

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, compiler simplification, application distribution, direct support of high-level paradigms and application interoperability [4]. These benefits of using abstract machines were firstly employed in an exclusive way. As an example, UNCOL [15] was designed to be used as the universal intermediate code of every compiler, but it was never employed as a distributed platform.

Java designers employed many benefits of using abstract machines in order to build a platform for delivering and running portable, dynamic and secure object-oriented applications on networked computer systems [10]. However, its design is focused on supporting the Java programming language, making it a difficult target for languages other than Java [12] (e.g. functional or logical applications).

Microsoft .NET platform is a system based on an abstract machine whose common language infrastructure (CLI, ECMA-335) has been designed from the ground up claiming to be a target for multiple languages, obtaining the previously mentioned benefits. Implementations of this abstract machine have been included in the new Windows Server 2003 family operating systems [25]. However, the design of .NET CLI and Java virtual machine have been performed in a monolithic manner, causing different drawbacks:

- Fixed computational model. There is a lack of an adaptability mechanism for the computing system. The abstract machine definition describes whether a garbage collector exits or not, but it does not offer the possibility to specify or adapt one. This way, real support to the C++ computational model could not be offered. It happens the same to other features like persistence or thread scheduling.
- Extensibility and heterogeneity. Instead of defining an extensibility technique capable of adapting the abstract machine to heterogeneous environments, Java and MS.NET defines different monolithic implementations depending on where the computing system will be deployed (JME, JSE and

JEE in Java's case, and .NET Compact Framework and .NET Framework for MS.NET).

- Ease of porting. Porting an abstract-machine based platform means making it work on a new operating system or/and hardware. In Java, not only the virtual machine implementation must be recompiled and adapted to the new platform, but also we need to port implementation of native methods. In the .NET Framework this port should be performed implementing the Platform Adaptation Layer (PAL) [14]. The PAL exposes a collection of 242 interfaces that provides an abstraction layer between the runtime and the operating system; these interfaces must be natively implemented on each target platform.
- Direct application interaction. Java and .NET virtual machine implementations execute a different virtual-machine instance to run each program. This is commonly justified by security reasons or system resources protection, which might be performed by untrusted malicious code or simply erroneous programs. The main drawback of this scheme is that different applications cannot directly interact with one another, being necessary to use a specific middleware or component architecture.

The main reason that causes these disadvantages might be security. As we have pointed out, a virtual machine should be a language-independent computing engine of the (operating) system —not just a specific language runtime. Following this point of view, security restrictions should be applied to application adaptation and interaction at the operating system level, in an integral way —but not restricting these features in the whole computing system.

In the design of our abstract machine, called nitrO, we have overcome the platform limitations pointed out by using reflection techniques [17]. The nitrO abstract machine offers an adaptable and extensible computing system, in which the virtual machine implementation is reduced to a minimum set of primitives. The platform can be extended by means of its own language, thus offering the possibility to adapt itself to heterogeneous environments —not needing to implement different platform versions. Great heterogeneity is offered by its reduced set of primitives and a high ease of porting is achieved by extending the platform on its own language, using reflection.

The rest of this paper is structured as follows. In the next section, we present the architecture of the system. The abstract machine design is presented in Section 3 and Section 4 shows an example of a scenario extending the platform with its own language. Subsequently, we describe specific concerns of our virtual machine implementation and how interaction with the platform is performed. Finally, we compare runtime performance and resource consumption with other well-known platforms, reporting the ending conclusions.

2 System Architecture

As we have pointed out in Section 1, adaptability, extensibility, heterogeneity, and application interaction are the main lack of most of current platforms based on abstract machines. We will now explain those principles followed in the nitrO abstract machine design, in order to avoid these deficiencies.

2.1 Minimum Set of Primitives

The design of the nitrO abstract machine has been performed trying to minimize the set of computational primitives used in its design, achieving two major benefits:

(1) Reducing the virtual machine implementation facilitates its deployment on heterogeneous systems. Therefore, devices with reduced computing and memory resources, as well as applications interested in interpreting nitrO abstract machine code (e.g. Internet browsers), could implement a virtual machine easily.
(2) If the abstract machine specification is undersized, porting its implementation to different operating systems will be straightforward.

2.2 Extensibility Mechanism

The reduced set of primitives gives the platform the benefits mentioned above, but the low-level abstraction of its programming-language makes it difficult to develop applications at a higher level of abstraction. Therefore, an extension mechanism should be offered to avoid this limitation.

We have identified and used reflection as the main technique to achieve system extensibility and adaptability. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [11]. The computational domain of a reflective system is enhanced by its own representation, offering its semantics and structure as computable data.

From the wide range of types of reflection [16], we have employed runtime introspection and structural reflection as a dynamic extensibility technique. Both have been employed to offer the platform the ability to extend itself on its own programming language. Basic and advanced computing features (from iteration loops or type systems to garbage collectors or persistence frameworks) can be developed using the reflective capabilities of nitrO. This capacity has two advantages:

(1) Any code implemented to extend the system is completely platform-independent —since it has been expressed in the abstract machine language.
(2) Its heterogeneous feature is not decreased. As an example, Java virtual machine has a native interface (Java Native Interface, JNI) that can be used to extend the platform (e.g., to modify the garbage collector). This native code is platform-dependent and, therefore, is part of the virtual machine implementation; otherwise, extending nitrO does not imply any virtual machine modification (i.e., the garbage collector can be implemented in its own language, or a new type can be added).

2.3 Object-Oriented Prototype-Based Computational Model

The computational model defined by the abstract machine will be used in the overall system, thus the abstraction level must be carefully specified:

- A high abstraction level would reduce the platform heterogeneity feature.
- A low abstraction level would make it difficult the interoperation of applications implemented in different programming languages —the programmer should be capable of easily accessing any application regardless of its programming language.

We have used the object-oriented paradigm to define the abstract-machine's computational-model, in order to facilitate program interaction. However, we have reduced the computation model to a minimum, suppressing the existence of classes. Thus we selected the object-oriented prototype-based model, in which the main abstraction is the object [1]. The use of this model provides different benefits to our platform:

(1) Suppressing the class abstraction simplifies the computational model, decreasing the size of the virtual machine and making its design more heterogeneous.
(2) 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 [23]. 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) is 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) is represented by *prototype* objects. This object has a set of initialized attributes that represent a common structure.

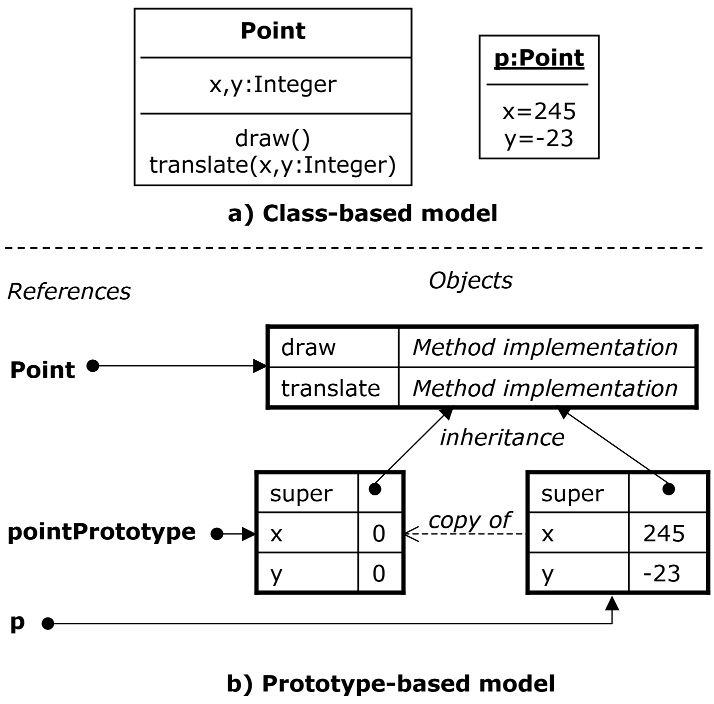


Fig. 1. Translation between the two object-oriented models.

- 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.

(3) Class-based languages like Java [9], Smalltalk [5] or Python [22] offer objects that represent classes at runtime (e.g., instances of Java `java.lang.Class` are objects representing classes). 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 [26].

(4) Our computational model has reflective capabilities. The dynamic adaptation of classes and objects structure by means of reflection causes the problem of scheme evolution [21]. How can an object's structure be modified without altering the rest of its class's instances?

Modeling this in class-based languages is complicated because its computational model does not fit well. This problem was detected in the development of MetaXa, a reflective Java platform implementation [6]. The solution they employed was a complex *shadow classes* mechanism where classes were duplicated when one of its instances was reflectively modified. One of the conclusions of their research was that prototype based languages express reflective features better than class-based ones: it is easy to derive an object from another object and change its fields or methods without affecting the original [6].

2.4 Adaptability

As the virtual machine is the substrate for the whole system, its computational features should be capable of being adapted depending on different requirements of heterogeneous environments. From basic language capabilities (such as operators, types or multiple inheritance), to complex computing features (like persistence, thread scheduling or garbage collection), the platform should offer the possibility to customize its features. Reflection is the main technique employed to achieve adaptability. It might be used at two levels:

(1) System level. Without changing the application code, structural and computational reflection offers system-level adaptability. As an example, following the Separation of Concerns principle [20,7], a persistence system can dynamically adapt system's behavior and structure making objects persist, not needing to modify the application code [19].
(2) Application level. Programs developed in the system are platform independent and can be executed in different environments, since the platform is highly heterogeneous. Therefore, applications must have a mechanism to dynamically inspect its environment and take decisions according to this information. Platform introspection (reflection) offers the applications this capability.

2.5 Direct Application Interaction

Java and .NET virtual machine implementations create different processes each time a program is run. One instance of the virtual machine program is executed per application. Justified by security reasons, this approach reduces their adaptability and extensibility features, for the reason that only an application may adapt or extend itself. Moreover, application interoperability has to be performed by using a middleware such as RMI.

The nitrO abstract machine and its implementations have been designed to run as a unique process of the operating system. This gives applications the ability to directly interact with any running program using reflection, whatever its source programming language might be.

We define the abstract machine as the system's language-independent computing engine. This way, security restrictions should be applied to application interaction at the operating system level, in an integral way —following techniques like access-control lists or the .NET code access security.

3 Abstract Machine Design

In this section, we introduce the design of the nitrO abstract machine. This design specifies the machine primitives, the programming language syntax, and its semantics; then, it will be explained how to use its extensibility and adaptability facilities. This design does not show the development of the virtual machine —i.e. an abstract machine implementation. Implementation matters will be presented in Section 6.

3.1 Object Concept

Objects are the basic abstraction of the abstract machine. At runtime, the system is composed of objects constituting different applications. The way a programmer should employ services offered by an object is by using a reference. Consequently, a reference is the unique mechanism to obtain and use an object.

The object concept is defined in a recursive way: an object is either a set of references to objects (members) or a primitive object. References between different objects may denote several meanings such as generalization, aggregation or composition. At runtime, the programmer may dynamically inspect and modify any object member by means of reflection primitives.

Following the scheme described in Section 2.3, the object concept, in addition to reflection primitives, are employed to describe every element of common class-based object-oriented languages [23].

3.2 Primitive Objects

Initially, the abstract machine offers a reduced set of primitive objects as well as the necessary references to access them. We use these objects and reflection to extend the platform abstraction level. There are four primitive objects (although it is not a must, we distinguish trait objects by naming them with a starting uppercase character):

(1) The `nil` object. This is the ancestor object —the root object of the inheritance tree. Its services will be inherited by every object; as a result, it holds the basic functionalities every object must offer (e.g. reflection routines).
(2) String objects. These objects represent information in our system; not only data but also computation —code that can be evaluated. Their behavior is grouped into the `String` trait object.

(3) The `Extern` object. This object offers the mechanism to locate and enhance the implementation of platform-dependent routines.
(4) The System object. It offers the capability of locating every existing object at runtime. It is a dynamic collection of objects. It is used to extend the platform (e.g., implementing a garbage collector or a persistence system.

3.2.1 The `nil` Object

As we have mentioned, it is the ancestor object. This primitive object is accessible throughout the platform by means of the `nil` reference. Like every object in the system, it has two member references:

- `id`: This is a reference to a string object that uniquely distinguishes the object identity. In this case, the id string value is `nil`.
- `super`: A reference to its super-object —the object from which the object is derived. As `nil` is the ancestor, this reference refers to itself.

3.2.2 String Objects

String objects represent information that can symbolize either data (arithmetic, logical or human-understandable information such as messages) or computation (code to be evaluated by the abstract machine). There is no difference in how information is represented; any kind of data or computation is characterized by string objects. The user should treat it in different ways depending on what she wants to represent.

Since string objects may represent code, and code statements might have nested statements inside, we represent this information with a context-free language instead of using a regular language —classical quoted strings would not be valid. We define `<` as the starting character of any string and `>` as the ending one. As they are different characters, strings can be nested.

Strings objects are automatically created when the programmer uses a reference to them —they are also automatically released when their last reference's scope finishes. A reference to a string object is represented by the string itself. Table 1 shows example string references denoting different type of meanings.

As we have explained above, every object has the `id` and `super` member references. In this case, every string object has its `super` reference pointing to the `String` trait object and its id reference to itself. Figure 2 shows an example of some existing objects snapshot at runtime. There are two string references: `<String>` and `<nil>`. Both have its super member pointing to the `String` trait object and their `id` reference pointing to themselves. There are

Table 1
Example String References and their meaning.

String Reference	Meaning
`<Hello world!>`	Human readable message (data)
`<true>`	Boolean constant (data)
`< return ← nil:<new>(); >`	Method or statement code (computation)
`<-2.5E+10>`	Real constant (data)

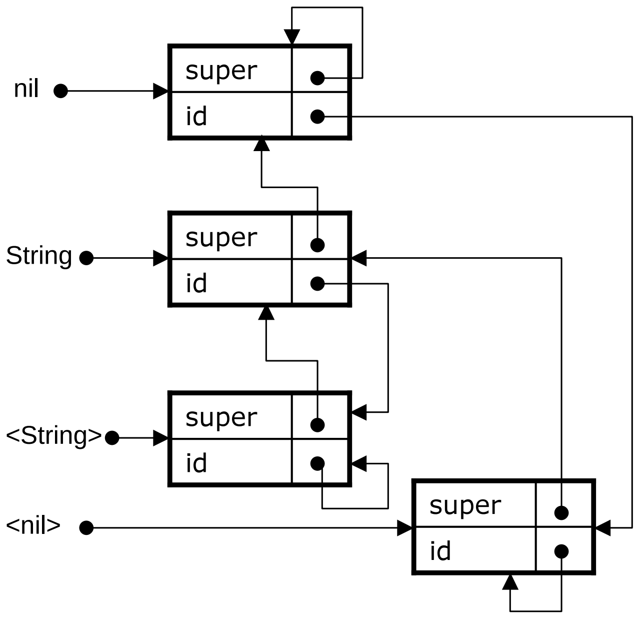


Fig. 2. Snapshot of existing objects and references at runtime.

two more references: `String` and `nil`. The objects they refer to have the id members pointing to `<String>` and `<nil>` respectively. `System` and `Extern` primitive objects have not been shown.

3.2.3 The `Extern` Object

We can classify system primitives into three groups:

- Computational primitives: These constitute the semantics of the abstract-machine programming-language, namely, the meaning of each programming language statement. Examples of this kind of primitives are member access or method invocation.
- Primitive objects: Objects that exist in the system when it starts up and offer specific functionalities. `String` and `nil` are both primitive objects.
- Operational primitives: Native primitive implementations that can be enhanced and modified, depending on specific platform requirements. These functionalities cannot be obtained from the two previous primitives groups,

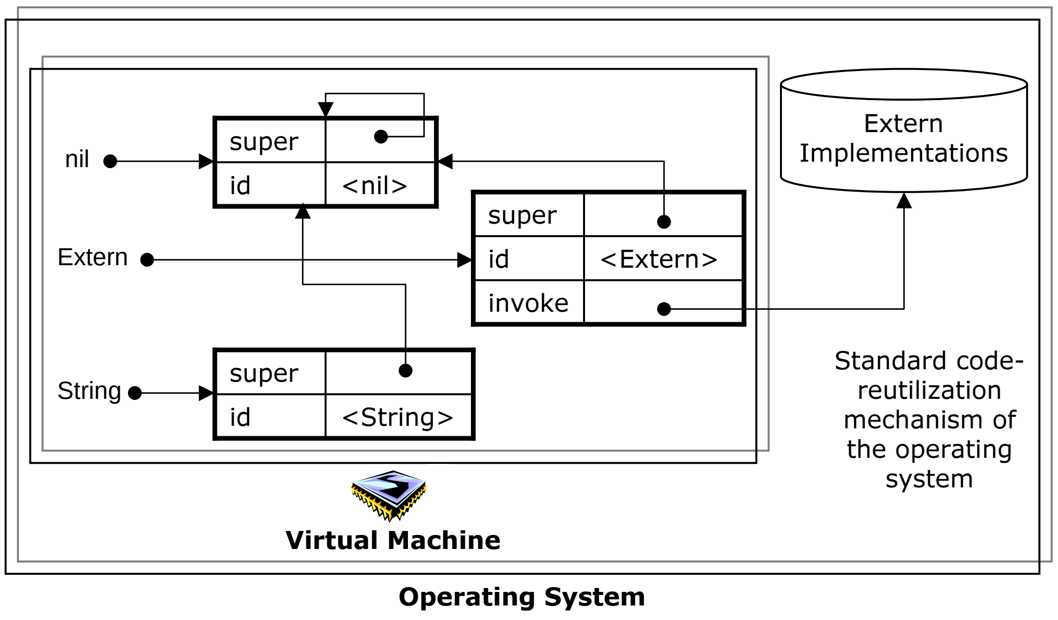


Fig. 3. Standard implementation of the operational primitives.

but are needed to develop specific programs or services.

We have designed the abstract machine taking into account that, instead of creating as many primitive objects as we might think necessary, we define the minimum required set in addition to a mechanism capable of enhancing them —that is what we call operational primitives. An example of two operational primitives is reading and writing to disk functionalities, needed to extent the platform with a persistence system.

The last primitive group, operational primitives, is located in the `Extern` object. As we will see, the `invoke` method of this object is responsible of executing the operational primitives. The implementation of operational primitives must be performed with a standard code-reutilization mechanism, which may depend on the operating system the virtual machine has been deployed in (e.g., a dynamic link library or a component architecture). Therefore, the modification of the operational primitives (machine-dependent code) will simply be carried out by the replacement of this module —the abstract machine implementation will not suffer any modification.

This platform-specific functionality implementation is quite similar to the one adopted by Java with its Java Native Interface. It is not necessary to modify the virtual machine implementation any time we need to add a new platform-specific primitive. Nevertheless, our approach places all the platform-specific code in a unique separate component, employing a standard code-reutilization mechanism -not spread out all over the library's native methods. Therefore, the grouping of platform-dependent code improves the porting of the platform's source code.

Figure 3 shows that the `id` and `super` members of the `Extern` object points to `<Extern>` and `nil` respectively. The `invoke` member, using a standard operating-system code-reutilization technique, access to specific operational primitives developed in the host operating system.

3.2.4 The `System` Object

The last primitive object is `System`. It has an `objects` member that refers to a collection of every existing object at runtime: it has as many members as objects are executing in the system. The name of each member is the respective unique object id.

3.3 Statements

We have seen the objects offered at the beginning. We are about to show how they could be used to program and extent the platform. The syntax we have employed is closed to Self [24], because of its simplicity and reflective capabilities. The nitrO language is also stack-based but uses dynamic scope as Lisp, APL or SNOBOL.

There are four basic statements:

3.3.1 Reference Creation

A reference is the sole mechanism to access objects. If the programmer wants to create a new reference in the stack, she just has to write the → token before the reference. These are two examples:

```
→ newReference ;
→ anotherReference ;
```

As we have mentioned, string references —and their respective objects— do not need to be explicitly created.

3.3.2 Reference Assignation

By means of the ← token, an existing reference could point to the object being referred by another one. The assignment direction is the one indicated by the arrow.

```
newReference ← anotherReference ;
```

3.3.3 Member Access

The programmer can access any object member by using a reference to an object. The syntax is `objectReference:memberId`. As we have explained, an object is a collection of references to other objects; `memberId` is a string reference that represents the member name. The following two sentences assign the `<nil>` string object to the `newReference`:

```
newReference ← nil:<id> ;
newReference ← <nil>:<id> ;
```

3.3.4 Code Evaluation

Our platform can dynamically generate code to be evaluated. This capability is used to extend and adapt the platform by means of its own language. The code to be evaluated is represented as string objects. Syntactically, the evaluation is denoted by a comma-separated list of references to parameters inside a pair of parenthesis. The evaluation could be performed synchronously or asynchronously, depending on the order of parenthesis.

```
→ code ;
code ← < → fooReference; > ;
code(newReference,anotherReference) ;
code)newReference,anotherReference( ;
```

The previous code creates a string object representing useless code and evaluates it synchronously and asynchronously respectively —passing two references as parameters. Asynchronous code evaluation cannot return any reference, but synchronous evaluation can.

Code evaluation may return a reference and receive parameters. Therefore, any code being evaluated synchronously has always two references: `params` and `return` —asynchronously evaluation does not have the `return` reference. What is assigned to the `return` reference at code execution is what the evaluation will give back. The `params` reference points to a numbered collection object: an object that collects the n-parameters in its `<0>`, `<1>` `<n-1>` members. The next example of code evaluation, returns its first parameter:

```
→ code ;
code ← < return←params:<0>; > ;
newReference ← code(anotherReference) ;
```

The result is that `newReference` points to the object referenced by `anotherReference`. In our platform, any kind of data and code information is represented by string objects. The programmer distinguishes code from data by evaluating the former. Following this point of view, method invocation is defined as the process of evaluating a string that is member of an object. The next statement stands for the `perimeter` method call of a `rectangle` object, passing a metric unit as a parameter:

```
result ← rectangle:<perimeter>(<cm>) ;
```

The method-evaluation mechanism offers an inheritance feature based on the super object member. If the programmer tries to evaluate a method that is not member of the object accessed, the system looks for the method on its super object. This process is performed in a recursive way, finishing when the super object is itself —i.e., the `nil` object. The abstract machine inheritance mechanism is dynamic, meaning that the `super` reference may change at runtime. Sometimes this feature is referred to as delegation [24]. The inheritance mechanism enhances the string evaluation semantics. Therefore, method code has one more reference than string evaluation: `sender`. This reference points to the object used in method invocation (the implicit object, the same as `this` or `self` in other languages). Because of inheritance, `sender` does not necessarily point to the object where the method is defined.

3.4 Methods of the `nil` Object

The basic functionalities every object offers are set as primitive members of the `nil` object (Table 2).

The `has`, `firstKey` and `nextKey` members give the programmer introspective capabilities. At runtime, dynamic information about object structure may be retrieved. Moreover, `set` and `remove` members offer structural reflection. Comparing to the class-based model, we would be able to dynamically modify class's attributes and methods. We could even modify the structure of only one of its instances.

The two last members provide explicit support for synchronization. Each object has a monitor associated with it, not defining any specific thread scheduling policy.

4 Extending the Platform

The platform abstraction level will be extended using reflection primitives, following the Smalltalk, Self and Forth scheme of loading an image file (footprint) at startup. As an example part of the image, we are going to create

Table 2
Methods of the `nil` object.

Method	Description
`new`	Creates a new object returning a reference to it. The object created has two members: `super` (pointing to `nil`) and its unique `id`. In order to prevent duplicated ids in other distributed virtual machines, we use the globally unique identifiers (GUID) specification of the OSF/DCE [13].
`delete`	Releases the implicit object.
`getRefCount`	Returns the number of exiting references to the implicit object.
`set`	Assigns a new member to the implicit object.
`has`	Returns whether the implicit object has a member or not.
`firstKey`	Returns a reference to the first lexicographically ordered member.
`nextKey`	Returns a reference to the next lexicographically ordered member.
`remove`	Removes the member reference, which has the same name as the parameter passed.
`enter`	The thread gains ownership of the `sender` monitor. If another thread already owns it, the current thread waits until it is unlocked.
`exit`	The thread releases the `sender` monitor. If there are threads blocked by this monitor, one of them may acquire it.

a trait `Object`, which groups any common behavior we think objects should share.

```
  → Object;
  Object ← nil:<new>();
```

If we want that every object (except `nil`) should inherit from `Object`, we can change the inheritance tree by means of structural reflection.

```
  String:<set>(<super>,Object);
  Extern:<set>(<super>,Object);
  System:<set>(<super>,Object);
```

The current problem is that, any time a new object is created, its parent will be `nil` —not `Object`. We may extend this behavior, implementing the next `newChild` method in `Object`:

```
Object:<set>(<newChild>,<
  return ← nil:<new>();
  return:<set>(<super>,sender);
  >);
```

Now, any time we want to create a derived object from another one, we just have to invoke the `newChild` method of the latter. Note that, using reflection, we could have taken the `nil:<new>` object and replace it with the new implementation —this is what we did in the implementation of garbage collectors [16].

As an example of adding native operational primitives, we will define string comparison:

```
String:<set>(<==>,<
  → param;
  param ← params:<0>;
  return ← Extern:<invoke>(<String>,<==>,sender,param);
  >);
```

The `invoke` member has four parameters: a string representing where the native method is located, the method name, the implicit object, and a method parameter. The virtual machine, any time the `invoke` method of the `Extern` object is called, executes a routine implemented in the standard reutilization-code mechanism explained in Section 3.2.3. The `<==>` external function shown above implements a native lexicographical string comparison.

At this time, we can compare any object with their ids:

```
Object:<set>(<==>,<
  → param;
  → paramID;
  → senderID;
  param ← params:<0>;
  paramID ← param:<id>;
  senderID ← sender:<id>;
  return ← paramID:<==>(senderID);
  >);
```

It is also possible to include logical information using the existing primitives:

```
→ Boolean;
Boolean ← Object:<newChild>();
→ true;
true ← Boolean:<newChild>();
→ false;
false ← Boolean:<newChild>();
true:<set>(<ifTrue>,<
  → trueCode;
  trueCode ← params:<0>;
```

```
   return ← trueCode();
   >);
true:<set>(<ifFalse>,<>);
false:<set>(<ifTrue>,<>);
false:<set>(<ifFalse>,<
   → falseCode;
   falseCode ← params:<0>;
   return ← falseCode();
   >);
Boolean:<set>(<if>,<
   → code;
   code ← params:<0>;
   return ← sender:<ifTrue>(code);
   >);
Boolean:<set>(<ifElse>,<
   → ifCode;
   → elseCode;
   ifCode ← params:<0>;
   elseCode ← params:<1>;
   sender:<ifTrue>(ifCode);
   sender:<ifFalse>(elseCode);
   >);
```

The code creates the trait object `Boolean` and their derived objects `true` and `false`. By means of polymorphism, we define the logical semantics of our specific platform. In this scenario, the code to be evaluated in the conditional clause is passed as a parameter. Next code shows an example of these conditional methods:

```
   → oneObject;
   oneObject ← Object:<newChild>();
   → bool;
   bool ← oneObject:<==>(oneObject);
   → result;
   result ← bool:<if>(< return ← <the same>; >);
```

Evaluating the previous code, the result reference points to the `<the same>` string object, because the comparison performed returns `true`.

Following this scheme, platform extension can be done easily. Boolean operators, arithmetic integer and real types —with its typical operations—, loops and object cloning that simulates class instantiation, are examples of what has been developed using the abstract-machine language itself.

We state that these are examples extensions because the platform can be built differently, following specific programmer requirements. For instance, new `Rational`, `Real` and `Integer` trait objects could be added to represent additional types of the language, and their typical operations could be developed as operational primitives. This type enhancement would produce a bigger but faster implementation —this was precisely the technique applied to define a typed Smalltalk, improving the typeless performance of Smalltalk-80 [2].

With the sample code shown, we demonstrate how the platform abstraction level may be enhanced with its own language. The consequence is that the user customizes the programming language, without modifying the virtual machine.

More advanced computational features such as garbage collection, persistence and distribution systems, as well as customizable thread schedulers have been developed using reflection [16,18]. An example garbage collector we have developed adapts the `new` member of the `nil` object replacing it with a new routine (using structural reflection). The recent `new` method, besides creating the object, counts the number of objects produced. Then, if this counter exceeds a customizable value, the garbage collector will be executed. By means of analyzing every runtime object (offered by `System`), the introspective analysis of object members and their reference counters (`getRefCount` member), inaccessible objects will be found and deleted [16]. More sophisticated implementations of garbage collectors, such as two-space compactors or generational ones [8] could be developed with different implementations of the `new` and `delete` operational primitives (implemented in C++, inside the `Extern` object).

5 Virtual Machine Implementation

In this section, we will briefly show different issues of a specific implementation of the abstract machine we have developed. It will also be presented the description of a simple porting process to a Linux platform we have performed, proving its heterogeneity feature. We will suppress UML designs and code concerns, focusing on deployment matters. More detailed information can be obtained in [16].

The virtual machine must be developed as a unique operating-system process, offering the illusion that another microprocessor exists. By using a standard inter-process communication protocol of the selected operating system, programmers will forward the application code they want to execute to the virtual machine, receiving the results after the evaluation. Any application can interoperate with the standard protocol selected, using seven messages: creating a new reference (`newReference`), getting an object's member (`getMember`) or evaluating members and strings both synchronously and asynchronously (`evaluateMember{Sync,Async}` and `evaluateString{Sync,Async}`).

We have first developed a virtual machine in C++, to be hosted in a Windows NT-based operating system as a system service. Then we have ported it to Linux (Section 5.1). When the computer turns on, the virtual machine process starts up and loads its footprint file `image.nitrO`. This image file execution produces the platform abstraction-level extension and configuration, depending on the specific system requirements.

Both the standard process-intercommunication protocol and code-reutilization mechanism of the operating system we have selected in our first implementation, were Microsoft COM. Therefore, any programming language (from basic scripting to advanced compiled ones), compiler or interpreter, may program and interact with the nitrO platform. The earliest applications we developed were a nitrO programming IDE developed in Visual Basic and a simple calculator in C#.

As an example, the next VBScript code interacts with the nitrO virtual machine service in order to execute an example program by means of the COM interface. This code can be executed in any application that uses the Windows Scripting Host, such as Internet Explorer or even Microsoft Office.

```
Dim vm, result, code
Set vm=CreateObject("nitrO.VM.1")
' Create references
vm.newReference("Person")
vm.newReference("mike")
vm.newReference("age")
' Creates a trait (class) Person
result = vm.evaluateMemberSync("Object", "<newChildWithId>",
              "<Person>", "Person")
' Trait (class) method: getAge
code = "<Person:<set>(<getAge>,<"
code = code + "return ← sender:<age>;"
code = code + ">);>"
result = vm.evaluateStringSync(code, "", "age")
' Trait (class) method: birthday
code = "<Person:<set>(<birthday>,<"
code = code + "newAge ← sender:<age>;"
code = code + "newAge ← newAge:<+>(<1>);"
code = code + "sender:<set>(<age>,newAge);"
code = code + ">);>"
result = vm.evaluateStringSync(code, "", "age")
```

```
' Creates an object (Person instance)
result = vm.evaluateMemberSync("Person", "<newChildWithId>",
             "<person>", "mike")
' Set age to 20
result = vm.evaluateStringSync("<mike:<set>(<age>,<20>);>","", "age")
result = vm.evaluateMemberSync("mike", "<getAge>", "", "age")
MsgBox ("Age before its birthday: " + result)
' Birthday
result = vm.evaluateMemberSync("mike", "<birthday>", "", "age")
result = vm.evaluateMemberSync("mike", "<getAge>", "", "age")
MsgBox ("Age after its birthday: " + result)
```

The previous code defines a COM reference to the nitrO virtual machine and creates three references using the `newReference` message. Then, it constructs a new child of `Object` (using the `evaluateMemberSync` message) and inserts the `getAge` and `birthday` methods (by means of `evaluateStringSync`). This object may represent a class because it does not offer a state, only behavior. Afterwards, we create an object with a specific `age`, representing a class instance. Its state is modified by the `birthday` method inherited. This way, we can program and interact with the system using any COM-compliant language.

Thinking in a testing environment, the nitrO virtual machine is also obtainable as an *instance per application* system (each program runs on a separate virtual machine instance, as Java does), protecting system resources as well as the rest of platform applications. Once a piece of code has been tested, it will be possible to introduce it in the system using the COM interface, or even be incorporated into the image file.

5.1 Linux Port

Once we developed and tested our implementation with Windows NT-based systems, we have ported it a to Debian 2.4.20 Linux distribution running in a HP NetServer with dual PII processor. Due to its heterogeneous design, the porting process has been done in a straightforward way. The following enumeration lists the different components of the nitrO platform and how they have been ported to Linux:

- The virtual machine. An ANSI/ISO C++ program consisting of 6,791 lines of code. This virtual machine component interprets the semantics of the reduced set of computational primitives. This port only required recompilation.

- Operational primitives placed in the `Extern` object. As we have mentioned in Section 3.2.3, this code should be developed using a standard operating-system code-reutilization mechanism. Although the semantics of each primitive does not change, the interaction technique requires a porting effort. We employed Microsoft COM in the Windows implementation –327 lines of code. This mechanism was substituted by 32 C++ lines that encapsulate the `Extern` object in a Linux dynamic linked library —also known as *shared objects*.
- The same happens with the virtual machine deployment. In the Windows implementation it was developed as a system service offering a COM interface. In the Linux port, we background the nitrO virtual machine process, allowing any client application interact with the platform by means of a pipe named *nitrOVMpipe*, a standard Inter-Process Communication (IPC) mechanism. This change involved the substitution of the 1,795 lines of code employed in Windows operating system for 274 lines to implement the named pipe IPC.
- Platform definition. The abstract machine has been designed to define its own specific platform using its programming language. Therefore, the main part of the system is completely platform-independent and do not need to be ported.

The source code, binaries of the two ports, sample applications, and UML design diagrams of these implementations can be downloaded from:

`http://http://www.di.uniovi.es/reflection/lab/`

6 Results

In order to compare different virtual machine implementations, we have selected —apart from the nitrO virtual machine— two well-known platforms: Java^{TM}from Sun Microsystems and Microsoft .NET. Looking for a rich comparison, we have tested two different editions of each platform: a small embedded-oriented implementation and a standard edition. These are the specific implementations tested:

- K Virtual Machine (KVM). The reference implementation of Sun's Connected Limited Device Configuration (CLDC) specification. CLDC is the foundation for the Java Micro Edition (JME) runtime environment targeted at small, resource-constrained devices, such as mobile phones, pagers, and mainstream Personal Digital Assistants (PDA).
- JSDK Java Virtual Machine (JVM). The reference implementation of Sun's Standard Edition of the Java Platform. It has been implemented using a hotspot Just in Time (JIT) compiler, which enables managed code to run

in the machine language of the target platform.

- Microsoft .Net Framework's Common Language Runtime (CLR). The professional Common Language Infrastructure (CLI, ECMA-335) implementation of Microsoft, released with the .Net Framework. As the JVM, this implementation employs a JIT native code generation.
- The mono interpreter (mint). Our first idea was to use the .Net Compact Framework edition of the Microsoft platform. The main problem was that the emulator of Windows CE and Pocket PC offered by Visual Studio 2003, uses the .Net Framework's CLR. Developing software for these two operating systems just means reducing the class library of the platform. Once the application has been tested on the desktop computer, a smart device development system deploys it on the target PDA or PocketPC with .Net Compact Framework.

  As we want to measure every implementation on the same computer we have rejected the MS Compact Framework, selecting the Mono open implementation of Microsoft .NET platform, on its interpreter release. Mint lacks of a JIT compiler, but is more suitable for resource-constrained devices.
- nitrO virtual machine. Our reflective heterogeneous and extensible virtual machine.
- nitrO virtual machine enhanced with native routines. As we are going to measure the virtual machine performance and its size, this virtual machine modification increases the number of native routines (located in the `Extern` object). Each part of the algorithms to be measured is developed in native code, making it faster and bigger, but much less portable. It will be compared with the two others JIT-based platforms in order to obtain an idea of future JIT implementations of the nitrO platform.

What has been compared at first is the size of each virtual machine implementation. Table 3 and Figure 4 show different sizes of the executables and the RAM employed to run a simple *Hello World* application on each platform. This measurement gives us an idea of which platform may be suitable for using in small devices (KVM, mint and nitrO editions of each platform). The rest of implementations offer bigger sizes, but better performance.

The nitrO virtual machine implementation is the smallest one (224 Kbytes) and it requires the least amount of memory to run a simple program. KVM is also a small one and needs 7 times less memory than the CLR to execute the same program.

Although native nitrO implementation is just one kilobyte bigger than the reflective nitrO, it is not significant: we have just implemented natively different parts of the specific programs to be measured. A commercial standard edition may implement much more routines on native code —or all of them if it incorporates a JIT compiler.

Table 3
Sizes of executable files.

Virtual Machine	Executable Size	RAM Employed
KVM	268 Kbytes	800 Kbytes
JVM	1,768 Kbytes	4,676 Kbytes
CLR	3,075 Kbytes	6,816 Kbytes
mint	1,859 Kbytes	3,142 Kbytes
nitrO	224 Kbytes	552 Kbytes
native nitrO	225 Kbytes	553 Kbytes

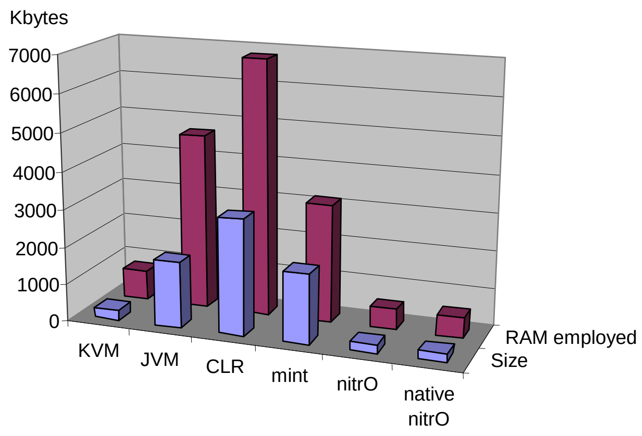


Fig. 4. Executable sizes and RAM employed.

Concerning to performance, we have used a selection of programs from the standard Java Grande benchmark suite [3]. The metric employed was execution time. All tests were carried out on a lightly loaded 1.0 GHz iPIII system with 256 Mb of RAM running WindowsXP.

We have selected a typical low-level operation (Loop) and two kernel benchmarks (LuFact and HeapSort) chosen to represent code likely to appear in any real application [3]. We have also implemented one of the representative real applications of the Java Grande Benchmark, in which any I/O and graphical component are removed [3]. By providing these different types of benchmarks, we could observe the behavior of most complex applications and interpret the results.

The metric employed was execution time: the wall clock time required to execute the portion of benchmark code that comprises the *interesting* computation. These are the codes employed:

- Loop. Low-level operation that measures loops overheads, for a simple for loop, a reverse for loop and a while loop. We have measured one million iterations of each kind of loop.

- LUFact. Kernel benchmark that solves a 100x100 linear system using LU factorization followed by a triangular solve. This is an evolution of the well-known Linpack benchmark. Memory and floating-point intensive.
- HeapSort. Classical kernel benchmark that sorts an array of 5,000 integers using a heap sort algorithm. It is memory and integer intensive.
- MolDyn. An application benchmark that models the behavior of 2,048 argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The solution is advanced for 100 timesteps. Performance is reported in units of timesteps per seconds.

We will first evaluate the low-level and kernel benchmarks separate from the MolDyn application. Since the memory employed in this application is much more extensive than the others, we will analyze it apart from them in Section 6.1 –the scales employed are extremely different.

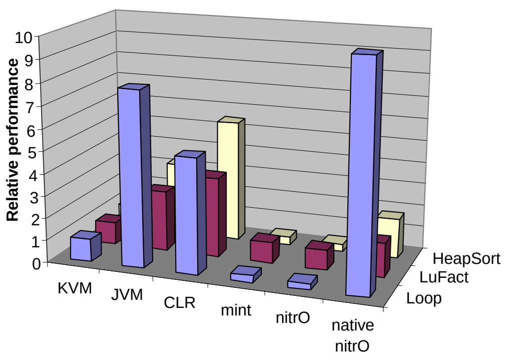
Figure 5 shows the averaged relative performance for each of the tested systems on the first three benchmarks used. The reference platform implementation (relative performance equals to 1) was the K Virtual Machine.

The most relevant feature of these results is that the standard implementations (JVM, CLR and native nitrO) give much higher performance that its resource-constrained editions (average of 4.6, 4.74, and 80.42 respectively). On the other hand, the evaluation of embedded systems, including nitrO, was similar to KVM: 0.93 (mint) and 0.88 (nitrO).

The native nitrO relative performance of the Loop benchmark has been cut down. Although its real value was 238 times faster than KVM, it has not been represented to make Figure 5 more legible. Despite the biggest average performance of the native nitrO platform, this way of implementing a virtual machine is the least portable. Every routine developed in native code must be ported and recompiled to every different target platform. Although JIT-based platforms are not generally faster than native code, they offer platform independency because translation to native code is performed at runtime —if native code performance were needed, then the C language would be a better choice than any virtual machine.

If we compare the size of the virtual machine with its relative performance, we may notice that the bigger its size is, the better performance is offered. Therefore, depending on the kind of platform we need (standard or resource-constrained) we may select one implementation or another. It is a trade-off between resource consumption and runtime performance.

To gauge how an implementation makes the most of its resources, we have also measured *relative performance by size*, which is the multiplication of the time employed to execute a benchmark by the amount of memory needed to run it (relative to the KVM platform). This measurement represents the runtime



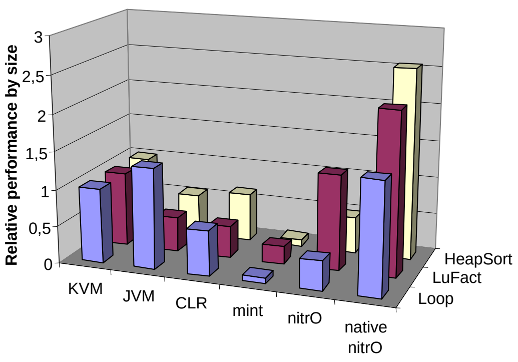


Fig. 5. Relative performance of KVM and relative performance by size of KVM.

performance relative to the resources used on each execution. Figure 5 shows the results.

The expected results were confirmed. This metric indicates that standard implementations are not faster than embedded ones, taking into account the memory employed to run each program. Native nitrO is the fastest (but it is not portable at all). Afterwards goes KVM and, with quite alike values, JVM (0.78) and nitrO (0.72). CLR and Mint implementations obtain 0.55 and 0.15 respectively.

The results obtained imply that the two Java platforms make better use of computing resources than .NET ones. Nitro virtual machine is in the middle, quite near to Java (0.81). However, we must also measure a qualitative feature: heterogeneity, the capability to adapt the computing engine to different computing environments, not needing to modify its implementation. Java platform has many routines implemented natively (they can also be enhanced by using JNI). In contrast, nitrO has an extremely reduced set of primitives and they are extended by means of reflection. This extension is expressed in its own language, and so, maximum portability of the platform is achieved.

6.1 Real Application Assessment

The Molecular Dynamics (MolDyn) application models N-body particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. Performance is reported in interactions per second. The number of particles is give by N. We have chosen N with the standard value of 2,048 [3]. The application has been coded in Java, C# and nitrO. The program consists of 3,360 lines of code written in the nitrO virtual machine language —1,057 and 1,037 lines in C# and Java, respectively.

Figure 6 shows the *performance* and *performance by size* measurements relative to the K virtual machine. It also illustrates the memory employed on

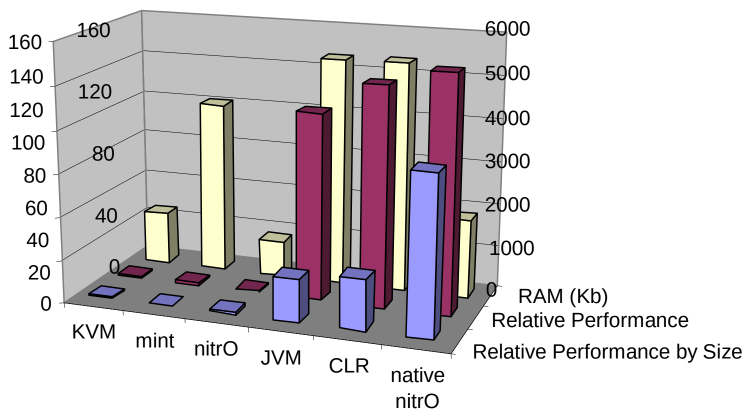


Fig. 6. Measurements of the MolDyn application execution.

each MolDyn application execution. Since the range of values of the two first rows (KVM relative values) is completely different from the RAM employed on each execution (Kbytes), we have separated both scales. Therefore, the values of the first two rows are shown on the left of the figure, whereas the last one appears on the right.

As happened with previous benchmarks, JIT-based platforms employ much more memory that the interpreter-based ones: the execution of the application in KVM requires a footprint of 1,236 Kbytes, while the same program run by the JVM implementation uses 5,325 Kbytes. At the same time, the three virtual machines that execute native code are much faster than the ones that interpret it –average relative performance of 89 vs. 1.14 in the standard and embedded Java implementations, respectively.

New results can be observed in this assessment. The *performance by size* metric shows that, opposite to to low-level and kernel analysis, standard implementations of the virtual machines make better use of systems resources than their resource-constrained counterparts. As an example, the execution of the MolDyn application over the standard JVM is more that 20 times faster than KVM, employing the same resources. The deduction of this second assessment is that, when the system has enough resources, the deployment of a virtual machine with a JIT is worth. Although a JIT compiler requires more systems memory, it makes better use of resources when it could be employed.

This time, the interpreter nitrO *performance by size* measurement is 1.43 times better than KVM and 3.3 in comparison with mint. This fact and the good results shown by the native nitrO implementation make us believe that future inclusion of a JIT compiler in the nitrO virtual machine will obtain good results. It could be deployed in systems that have enough resources to run it.

After this evaluation report, we can conclude that reflection offers high heterogeneity, platform portability, and good exploitation of computing resources.

If better performance is needed and there are enough system resources, the most suitable technique (employing more resources, but not loosing platform portability) would be a good JIT compiler. At the same time, our platform can be used as a 224 Kbytes implementation for small resource-constrained devices.

7 Conclusions

Abstract machines have been widely employed to design programming languages because of the many advantages they offer. Different examples of such benefits are code portability, compiler simplification, interoperability, distribution and direct support of specific paradigms. Although performance was the main drawback of employing virtual machines in the past, modern techniques like adaptive (hotspot) Just In Time compilation have overcome this weakness. Nowadays, well-know abstract-machine-based platforms such as Java$^{\text{TM}}$or Microsoft .NET are commercially used.

We have pointed out that the main limitations of existing abstract machines are caused by their monolithic designs. Most of existing designs are focused on supporting a specific programming language, being difficult to adapt the platform to different computing models. Therefore, common lacks of current abstract machines are heterogeneity, extensibility, adaptability, platform porting, and direct application interaction.

The computational model employed to design our abstract machine, called nitrO, has been the prototype-based object-oriented one. In order to facilitate language-independent program interaction, we have chosen the most reduced object-oriented computational model. Different works have demonstrated that any abstraction modeled with a class-based object-oriented language can be easily and intuitively translated into the prototype-based computational model [23,26]. It is also more suitable in reflective environments [6].

Reflection has been the key technique selected to define our flexible abstract machine, overcoming the previously mentioned drawbacks. By using its own language, reflection gives the platform extensibility, adaptability and application interaction features, being capable of deploying it in extremely heterogeneous environments. Different programming language features could be customized without modifying the virtual machine implementation.

We first select a reduced set of primitives that offers the basic group of computing routines needed to develop any program. The resulting small size (224 Kbytes) makes it very suitable for employing it on resource-constrained devices. However, the compact set of primitives offers a very limited computing platform. By means of reflection, the abstraction level could be increased on its own language, not needing to modify the virtual machine implementation, nor deploying different platform editions. As an example, we have shown the extension of computing features such as a Boolean algebra or objects comparison; more advanced features, such as an adaptable garbage collector framework, have also been developed.

The extension of the abstraction level, expressed in its own language, facilitates the port of the whole platform to new target systems. Therefore, the migration of the Windows implementation to a Linux system has been performed in a straightforward way: the necessary changes were just the encapsulation of the `Extern` object and the selection of an inter-process communication protocol.

We have also confirmed that our reflective implementation makes good use of computing resources. However, runtime performance is the typical drawback of developing a virtual machine as an interpreter. When we have enough resources and better performance is needed, a JIT compiler is the better alternative to execute applications in a faster way, without loosing the benefits of using reflection. Future work will be directed at developing a JIT compiler for the nitrO platform.

As a result, we identify reflection as a technique to dynamically extend and adapt a platform without modifying the virtual machine implementation, and not loosing portability of the whole system. This makes a system very heterogeneous, being able to use it in different environments —from small resource-constrained devices or embedded in different applications, to full-featured application servers making use of a JIT compiler.

References

[1] A.H. Borning: *Classes Versus Prototypes in Object-Oriented Languages*, Proceedings of the ACM/IEEE Fall Joint Computer Conference 36-40, 1986.

[2] G. Bracha, and D. Griswold: *Strongtalk: typechecking Smalltalk in a production environment*, ACM Symposium on Object-Oriented Programing: Languages, Systems, and Applications (OOPSLA), ACM SIGPLAN Notices 28 (10), 1993.

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

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

[5] A. Goldberg, and D. Robson: Smalltalk-80, the Language and its implementation, Addison-Wesley, Reading, MA, 1983.

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

[7] W.L. Hürsch, and V. Lopez: Separation of Concerns, Technical Report UN-CCS-95-03, Northeastern University, Boston, USA, 1995.

[8] R. Jones, and R. D. Lins: Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, New York, 1996.

[9] B. Joy, G. Steele, J. Gosling, and G. Bracha: Java Language Specification, Addison-Wesley, 2000.

[10] T. Lindholm, and F. Yellin: The Java Virtual Machine Specification, Addison Wesley, Reading, MA, 1996.

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

[12] K. Meijer, and J. Gough: Technical Overview of the Common Language Runtime, Microsoft .Net white papers, URL: http://docs.msdnaa.net/ark/Webfiles/whitepapers.htm.

[13] S. Miller: DEC/HP Network Computing Architecture Remote Procedure Call Run Time Extensions version OSF TX1.0.11, Open Software Foundation, Cambridge, MA, 1992.

[14] T. Neward, D. Stutz, and G. Shilling: Shared Source CLI Essentials, O'Reilly & Associates, 2003.

[15] K.V Nori, U. Ammann, and K. Jensen: Pascal - the Language and its Implementation, Barron, D.W. (ed) Wiley, Chichester, England, 1981.

[16] F. Ortin: A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine, PhD Thesis, University of Oviedo, Spain, 2002.

[17] F. Ortin, and J.M. Cueva: *Implementing a Real Computational-Environment Jump in order to Develop a Runtime-Adaptable Reflective Platform*, ACM SIGPLAN Notices 37 (8), 2002.

[18] F. Ortin, and J.M. Cueva: *Dynamic Adaptation of Application Aspects*, Elsevier Journal of Systems and Software 71(3), 2004.

[19] F. Ortin, A.B. Martinez, D. Alvarez, and J.M. Cueva: *A Reflective Persistence Middleware over an Object Oriented Database Engine*, XIV Brazilian Symposium on Databases (SBBD), Florianopolis, Brasil, 1999.

[20] D.L. Parnas: *On the Criteria to be used in decomposing systems into modules*, Communications of the ACM, Vol. 15, No. 12, 1972.

[21] J. Roddick: *A Survey of Schema Versioning Issues for Database Systems*, Information and Software Technology, 37, 1995.

[22] G. Rossum, Python Reference Manual, iUniverse.com, 2000.

[23] D. Ungar, D. Chambers, B.W. Chang, and U. Hölzl: Organizing Programs without Classes, Lisp and Symbolic Computation, Kluwer Academic Publishers, 1991.

[24] D. Ungar, and R.B. Smith: *SELF: The power of simplicity*, ACM Symposium on Object-Oriented Programming: Languages, Systems, and Applications (OOPSLA), ACM SIPLAN Notices 22 (12), 1986.

[25] J. Whitney: Windows .NET Server: 7 New Features for Developers, DevX, 2002, http://www.devx.com/SummitDays/Article/10092.

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