LANGUAGE-NEUTRAL SUPPORT OF DYNAMIC INHERITANCE

Jose Manuel Redondo, Francisco Ortin
University of Oviedo, Computer Science Department,Calvo Sotelo s/n, 33007, Oviedo, Spain

J. Baltasar Garcia Perez-Schofield
University of Vigo, Computer Science Department, As Lagoas s/n, 32004, Ourense, Spain

Keywords: Dynamic inheritance, structural reflection, dynamic languages, JIT compilation, SSCLI, virtual machine, prototype-based object-oriented model.

Abstract: Virtual machines have been successfully applied in diverse scenarios to obtain several benefits. Application interoperability and distribution, code portability, and improving the runtime performance of programs are examples of these benefits. Techniques like JIT compilation have improved virtual machine runtime performance, becoming an adequate alternative to develop different types of software products. We have extended a production JIT-based virtual machine so they offer low-level support for structural reflection, in order to obtain the aforementioned advantages in dynamic languages implementation.
As various dynamic languages offer support for dynamic inheritance, the next step in our research work is to enable this support in the aforementioned JIT-based virtual machine. Our approach enables dynamic inheritance in a language-neutral way, supporting both static and dynamic languages, so no language specification have to be modified to enable these features. It also enables static and dynamic languages to interoperate, since both types are now low-level supported by our machine.

1 INTRODUCTION

Dynamic languages like Python (Rossum and Drake, 2003) or Ruby (Thomas et al., 2004) are frequently used nowadays to develop different kinds of applications, such as adaptable and adaptive software, Web development (the *Ruby on Rails* framework (Thomas et al., 2005)), application frameworks (*JSR 223* (Grogan, 2008)), persistence (Ortin et al., 2004) or dynamic aspect-oriented programming (Ortin and Cueva, 2004). These languages build on the Smalltalk idea of supporting reasoning about (and customizing) program structure, behavior and environment at runtime. This is commonly referred to as *the revival of dynamic languages* (Nierstrasz et al., 2005).

The main objective of dynamic languages is to model the dynamicity that is sometimes required in building high context-dependent software, due to the mobility of both the software itself and its users. For that reason, features like meta-programming, reflection, mobility or dynamic reconfiguration and distribution are supported by these languages. However, supporting these features cause two main drawbacks: the lack of static type checking and a considerable runtime performance penalty.

Our past work (Ortin et al., 2005) (Redondo et al., 2008) (Redondo et al., 2006a) (Redondo et al., 2006b) has been focused on applying the same approach that made virtual machines a valid alternative to develop commercial software. Since many virtual machines of dynamic languages are developed as interpreters, we have used an efficient virtual machine JIT compiler to evaluate whether it is a suitable technique to achieve advantages like improving their runtime performance. The chosen virtual machine was the Microsoft Shared Source implementation of the .Net platform (SSCLI).

Nowadays, there are initiatives to support dynamic languages modifying a production JIT-based virtual machine, such as (Rose, 2008). Our approach also adds new features to an existing virtual machine, but we want to introduce full low-level support for the whole set of reflective primitives that support the dynamicity of these languages, as part of the machine services. The main advantages of our approach are:

1. Language processors of dynamic languages can be implemented using the modified machine services. Low-level support of structural reflection

Manuel Redondo J., Ortin F. and Baltasar Garcia Perez-Schofield J. (2008).
LANGUAGE-NEUTRAL SUPPORT OF DYNAMIC INHERITANCE.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 5-12
DOI: 10.5220/0001870400050012

eases the implementation of its dynamic features.

2. Full backwards compatibility with legacy code. Instead of modifying the syntax of the virtual machine intermediate language, we extended the semantics of several instructions.
3. Reflective primitives are offered to any present or future language (they are language-neutral).
4. Interoperability is now possible between static and dynamic languages, since the machine supports both of them.
5. Obtain a performance advantage, since it is not necessary to generate extra code to simulate dynamic functionalities –see section 2.2.

Existing implementations do not offer all the advantages and features of our approach. For example, the Da Vinci virtual machine (OpenJDK, 2008) (prototype implementation of the JSR292 specification (Rose, 2008)) will likely add new instructions to its intermediate language. By doing this, its new features will not be backward compatible, since compilers must be aware of these instructions to benefit from the extended support they provide. Another example is the Microsoft DLR (Chiles, 2008), which aims to implement on top of the CLR several features to support dynamic languages. Languages must be designed to target the DLR specifically to benefit from its features, so this implementation is not language-neutral.

In our past work we had successfully implemented most of these primitives into the machine, enabling low-level support to add, modify or delete any object or class member. Prototype-based object-oriented semantics (Borning, 1986) were also appropriately introduced to solve those cases in which the existing class-based model had not enough flexibility to implement the desired functionality. The original machine object model evolved into a hybrid model that support both kinds of languages without breaking backwards compatibility, also allowing direct interoperability between static and dynamic languages (Redondo et al., 2008). The research prototype that incorporates all these dynamic features is named Reflective Rotor or ЯRotor.

In this paper we will explain the next step of our work, designing and implementing a new reflective primitive into our system which will enable the users to effectively use single dynamic inheritance over the hybrid object model that ЯRotor now supports. Languages that support this feature (such as Python (Rossum and Drake, 2003)) make possible, for example, to change the inheritance hierarchy of a class at run time (Lucas et al., 1995), allowing a greater degree of flexibility to its programs. With the addition of this new primitive, the full set of structural reflective primitives will be low-level supported by ЯRotor.

The rest of this paper is structured as follows. Section 2 describes the background of our work, explaining several concepts involved in its development and the current state of our implementation prototype. Section 3 details the design of the dynamic inheritance support over the existing machine. Section 4 presents details about the implementation of our design and finally section 5 describe the final conclusions and future lines of work.

2 BACKGROUND

2.1 Type Systems of Dynamic Languages

Dynamic languages use dynamic type systems to enable runtime adaptability of its programs. However, a static type system offers the programmer the advantage of early detection of type errors, making possible to fix them immediately rather than discovering them at runtime or even after the program has been deployed (Pierce, 2002). Another drawback of dynamic type checking is low runtime performance, discussed in the following section. There are some research works that try to partially amend the disadvantages of not being able to perform static type checking. These include approaches like integrating unit testing facilities and suites with dynamic languages (MetaSlash, 2008) or allowing static and dynamic typing in the same language (Meijer and Drayton, 2004) (Ortin, 2008). This is the reason why we have focused our efforts in improving runtime performance.

2.2 Runtime Performance of Dynamic Languages

Looking for code mobility, portability, and distribution facilities, dynamic languages are usually compiled to the intermediate language of an abstract machine. Since its computational model offers dynamic modification of its structure and code generation at runtime, existing virtual machines of dynamic languages are commonly implemented as interpreters. This fact, plus the runtime type checking additional cost, cause an important performance penalty when compared to “static” languages.

Since the research in *customized dynamic compilation* applied to the Self programming language (Chambers and Ungar, 1989), virtual machine implementations have become faster by optimizing the bi-

nary code generated at run time using different techniques. An example is the dynamic adaptive *HotSpot* optimizer compilers. Speeding up the application execution of dynamic languages by using JIT compilation facilitates their inclusion in commercial development environments.

Most works that use virtual machine JIT compilers to improve runtime performance of dynamic languages are restricted to compilers that generate Java or .Net bytecodes. These machines do not support structural reflection, as they were created to support static languages. Generating extra code is then needed to simulate dynamic features over these machines, leading to a poor runtime performance (Udell, 2003). Examples of this approach are *Python for .Net* from the Zope Community, *IronPython* from Microsoft or *Jython* for the JVM platform.

Our approach uses a virtual machine with JIT compilation to directly support any dynamic language. Unlike existing implementations, we extended the static computational model of an efficient virtual machine, adding the reflective services of dynamic languages. This new computational model is then dynamically translated into the native code of a specific platform using the JIT compiler. Instead of generating extra code to simulate the computational model of dynamic languages, the virtual machine supports these services directly. As a result, a significant performance improvement is achieved -see section 2.4.

2.3 Structural Reflection

Reflection is *the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions* (Maes, 1987). In a reflective language, the computational domain is enhanced with its own representation, offering its structure and semantics as computable data at runtime. Reflection has been recognized as a suitable tool to aid the dynamic evolution of running systems, being the primary technique to obtain the meta-programming, adaptiveness, and dynamic reconfiguration features of dynamic languages (Cazzola et al., 2004).

The main criterion to categorize runtime reflective systems is taking into consideration what can be reflected. According to that, three levels of reflection can be identified: *Introspection* (read the program structure), *Structural Reflection* (modify the system structure, adding fields or methods to objects or classes, reflecting the changes at runtime) and *Computational (Behavioral) Reflection* (System semantics can be modified, changing runtime behavior of programs).

2.4 ЯRotor

Compiling languages to the intermediate code of a virtual machine offers many benefits, such as platform neutrality or application interoperability (Diehl et al., 2000). In addition, compiling languages to a lower abstraction level virtual machine improves runtime performance in comparison with direct interpretation of programs.

Therefore, we have used the Microsoft .Net platform as the targeted virtual machine to benefit from its design, focused on supporting a wide number of languages (Meijer and Gough, 2000) (Singer, 2003). The selection of an specific distribution was based on the necessity of an open source implementation to extend its semantics and an efficient JIT compiler. The SSCLI (Shared Source Common Language Infrastructure, aka Rotor) implementation has been our final choice because it is nearer to the commercial virtual machine implementation: the Common Language Runtime (CLR) (Stutz et al., 2003).

To date, we have successfully extended the execution environment to adapt the semantics of the abstract machine, obtaining a new reflective computational model that is backward compatible with existing programs. We have also extended the Base Class Library (BCL) to add new structural reflection primitives (add, remove and modify members of any object or class in the system), instead of defining new IL statements. We refer to this new platform as Reflective Rotor or ЯRotor.

The assessment of our current ЯRotor implementation has shown us that our approach is the fastest if compared to widely used dynamic languages such as Python and Ruby (Redondo et al., 2008). When running reflective tests, our system is on average almost 3 times faster than the fastest commercial dynamic languages implementations. When running static code, our system is 3 times faster than the rest of tested implementations, needing a significantly lower memory usage increment (at most 102% more). Finally, we have also evaluated the cost of our enhancements. When running real applications that do not use reflection at all, empirical results show that the performance cost is generally below 50%, using 4% more memory.

2.5 The Object Model of ЯRotor

The current object model of our machine is the result of the needed evolution of the original SSCLI class-based model to give proper support to the reflective primitives. There are some inconsistencies between the class-based model and structural reflection. As stated in the MetaXa project (Kleinder and

Golm, 1996), a single object structure is very difficult to modify without altering the rest of its class instances (a class must reflect the structure of all its instances). This fact causes several problems (maintaining the class data consistency, class identity, using class objects in the code,...) involving a really complex and difficult to manage implementation (Golm and Kleinder, 1997). This is why we introduced the prototype-based object-oriented model into ЯROTOR.

In the prototype-based object-oriented computational model the main abstraction is the object, suppressing the existence of classes (Borning, 1986). Although this computational model is simpler than the one based on classes, any class-based program can be translated into the prototype-based model (Ungar et al., 1991). In fact, this model has been considered as a universal substrate for object-oriented languages (Wolczko et al., 1996).

For our project, the most important feature of the prototype-based object-oriented computational model is that it models structural reflection primitives in a consistent and coherent way. Dynamic languages use this model for the same reason. Modifying the structure (fields and methods) of a single object is performed directly, because any object maintains its own structure and even its specialized behavior. Shared behavior could be placed in the so called trait objects, so its customization implies the adaptation of types.

Although the so called *Common Language Infrastructure* (CLI) tries to support a wide set of languages, the .Net platform only offers a class-based object-oriented model optimized to execute "static" languages. In order to allow prototype-based dynamic languages to be interoperable with any existing .Net language or application, and to maintain backward compatibility, we supported both models. This way we can run static class-based .Net applications and dynamic reflective programs. .Net compilers could then select services of the appropriate model depending on the language being compiled. Consequently, compilers for a wider range of languages could be implemented. These compilers will generate the intermediate code of our machine, that is in fact syntactically the same as the original machine (to maintain backward compatibility). Examples of these languages are:

- Class-based languages with static typing (C#).
- Prototype-based languages with dynamic typing (e.g. Python (Rossum and Drake, 2003)).
- Class-based languages with static and dynamic typing (e.g. Boo (CodeHaus, 2008) or StaDyn (Ortin, 2008)).
- Prototype-based languages with static typing (e.g. StrongTalk (Bracha and Griswold, 1993)).

2.6 Dynamic Inheritance

When passing messages to a particular object, conventional class-based languages use a concatenation-based inheritance strategy. This means that all members (either derived or owned) of a particular class must be included in the internal structure of this class. Using this approach enables compile-time type checking. This will prove that there can never be an error derived from invoking a non existing message (Ernst, 1999), but at the expense of flexibility. In contrast, dynamic languages use delegation-based inheritance mechanism, iterating over the hierarchy of an object searching for the intended member. Since our system introduced prototype-based semantics, we also implemented a delegation-based inheritance mechanism to be used together.

Delegation - based inheritance is complemented with dynamic inheritance. In contrast with conventional class-based languages, prototype-based languages allow a inheritance hierarchy to be changed at run time (Lucas et al., 1995). More specifically, dynamic inheritance refers to the ability to add, change or delete base classes from any class at run time. It also includes the ability to dynamically change the type of any instance. This results in a much more flexible approach, allowing objects and classes of any program to better adapt to changing requirements. This type of inheritance is implemented by languages such as Python (Rossum and Drake, 2003), Self (Chambers and Ungar, 1989), Kevo (Taivalsaari, 1992), Slate (Project, 2008) or AmbientTalk (Cutsem et al., 2007).

Therefore, it is necessary to create the means to provide adequate support for dynamic inheritance to give a complete support of the dynamic features of dynamic languages.

3 DESIGN

In this section we will analyze the semantics of dynamic inheritance when applied over the object model of ЯROTOR. Therefore, we must take into account both class-based and prototype-based semantics. It is important to state that, while either models are supported, they will not be both present at the same time. Languages could be either class-based or prototype-based, but they will not use both models together. The implementation of dynamic inheritance will be done creating a new *setSuper* primitive, which will be

added to the already existing ones. We consider two main operations over each object model:

1. **Instance Type Change** (*setSuper* applied over an object). Substituting the current type of an instance with another one, performing the appropriate changes on its structure to match the new type.
2. **Class Inheritance Tree Change** (*setSuper* applied over a class). Substituting the base type of a class with another one, performing the necessary changes over the class hierarchy and instances.

In order to give a precise description, we will formalize the behavior of the new *setSuper* primitive. We assume that:

- C_a is the attribute set of class C.
- C_m is the method set of class C.
- C_p is the member set of class C ($C_p = C_a \cup C_m$)
- $C_a^+ = C_a \cup D_a^+$, $\forall$ D superclass of C. Represents the full set of attributes (including inherited) of C.
- $C_m^+ = C_m \cup D_m^+$, $\forall$ D superclass of C. Represents the full set of methods (including inherited) of C.
- $C_p^+ = C_a^+ \cup C_m^+$.

The described formalizations have been designed without breaking the restrictions of the object model over which they are applied. This design also relies on the reflective primitives that were implemented in our previous work –see section 2.4. Each formalization is given an example to clarify its functionality. All the given examples will refer to the class diagram shown in figure 1.

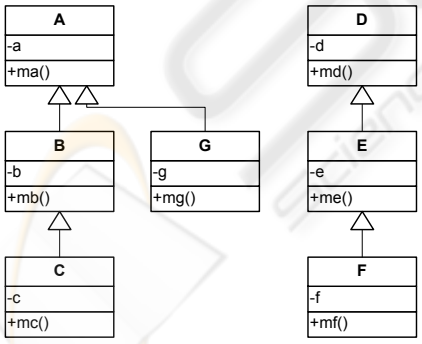

Figure 1: Example class diagram.

3.1 Class-based Model

The design of an **instance type change** is formalized as follows. Figure 2 shows an example of this operation:

- Let X, Y be classes and *o: X*.
- The *setSuper(o, Y)* primitive call modifies o structure this way:
  - Delete from o the member set D ($D = X_p^+ - (X_p^+ \cap Y_p^+)$)
  - Add to o the member set A ($A = Y_p^+ - X_p^+$)

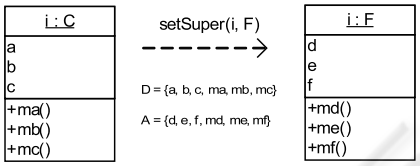

Figure 2: Type change (class-based model).

The design of an **inheritance tree change** of a class is formalized as follows. Figure 3 shows an example of this operation:

- Let X, Y be classes. Let Z be the base class of X.
- The *setSuper(X, Y)* primitive call modifies class X structure this way:
  - Delete from X the member set D ($D = Z_p^+ - (Z_p^+ \cap Y_p^+)$)
  - Add to X the member set A ($A = Y_p^+ - Z_p^+$)

It should also be noted that instance members of modified classes are dynamically updated when they are about to be used (lazy mechanism). The existing primitives already use this mechanism (Redondo et al., 2008), so we will take advantage of it. This is much faster than actively modifying all instances when a change is performed, specially if a large number of them are present.

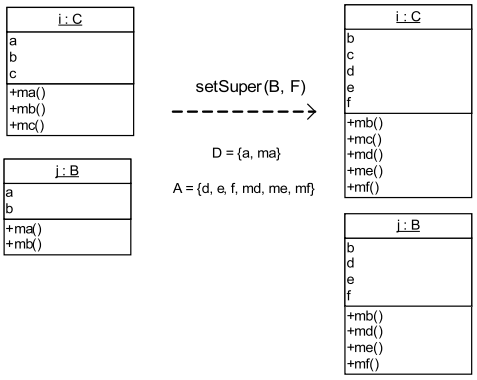

Figure 3: Inheritance tree change (class-based model).

3.2 Prototype-based Model

The design of an **instance type change** is formalized as follows. Figure 4 shows an example of this operation:

- Let *X*, *Y* be classes and *o: X*.
- The *setSuper(o, Y)* primitive call modifies *o* structure this way:
  - Delete from *o* the method set *D* ($D = X_m^+ - (X_m^+ \cap Y_m^+)$)
  - Add to *o* the method set *A* ($A = Y_m^+ - X_m^+$)

It should also be noted that existing attributes are always maintained in the instances (no attribute is added nor deleted).

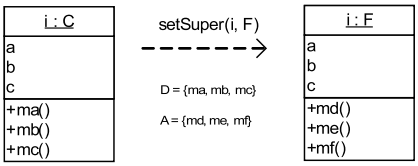

Figure 4: Type change (prototype-based model).

The design of an **inheritance tree change** of a class is formalized as follows. Figure 5 shows an example of this operation:

- Let *X*, *Y* be classes. Let *Z* be the base class of X.
- The *setSuper(X, Y)* primitive call modifies class *X* structure this way:
  - Delete from *X* the method set *D* ($D = Z_m^+ - (Z_m^+ \cap Y_m^+)$)
  - Add to *X* the method set *A* ($A = Y_m^+ - Z_m^+$)

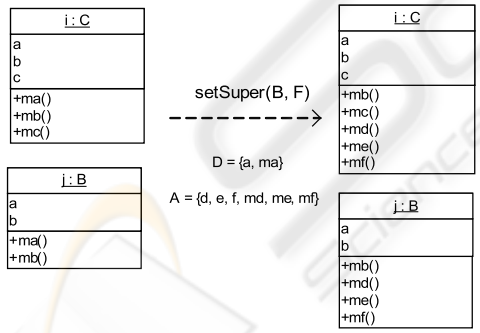

Figure 5: Inheritance tree change (prototype-based model).

The only difference between both models formalization is that the prototype-based model maintain instance attributes. Finally, it must be remarked that all operations will comply the following assumptions:

- The computational complexity of any operation is *O(n)*, being *n* the number of classes present in the involved classes (*X*, *Y*) hierarchies. The algorithm goes through each class once to perform the requested operation.
- In ЯRotor, when we ask via introspection for the type of an object or the base type of a class, it must respond with the dynamically assigned type.
- Adding a member to a set that already contains it has no semantics (no computation is performed).
- When new attributes are incorporated to instances or classes as a consequence of any of the described operations, default values are assigned according to the value that was assigned to them in its original declaration.

4 IMPLEMENTATION

The implementation could be divided in three important aspects. We will describe the techniques followed in each part to implement the described design.

4.1 The *setSuper* Primitive Interface

As we made with the primitives to add, modify and remove members, we changed our *NativeStructural* BCL class to allow access to the new *setSuper* primitive. The virtual machine core is then extended to incorporate this new service, linking the primitive call interface to its low-level implementation (Redondo et al., 2008). Therefore, our new extension naturally integrates with the already created infrastructure, being part of the virtual machine services. The simplest way to distinguish what object model is going to be used is adding a third boolean parameter to the primitive. This way, a user can easily choose either model and the system will select the appropriate behavior.

4.2 Instance Manipulation

One of the most challenging tasks in this work is how to effectively translate the designed dynamic inheritance model into the machine internals. The new *setSuper* primitive will take advantage of the primitives added in our previous work to successfully implement its features. However, these primitives never needed to modify parent-child or instance-class relationships, so this concrete part of the system must be studied. The capability to add or remove members to any object or class in the system alone fell short to implement the desired functionality. Although the changed entity could have the exact interface we pretend, its internal type will not be correct.

So, in order to achieve an instance type change, we have to carefully review the low-level infrastructure representation of instances (Stutz et al., 2003). This way, we found that every instance is bound to its

class using a pointer to a unique class: the *MethodTable*. We found that dynamically changing an instance method table produces the desired effect, and the system responds as expected when types of instances are dynamically requested.

Therefore, by appropriately combining our existing structural reflection primitives with a *MethodTable* substitution of the involved instance we can achieve the desired type change functionality. Once this is solved, it was easy to compute the *A* and *D* member sets and apply them to implement the described model design using the primitives of our previous work (Redondo et al., 2008).

4.3 Class Manipulation

Although in the internal structure of a class there exist a pointer to its base type, dynamically changing this pointer to the desired class would not work. This is because the original *SSCLI* virtual machine implements introspection routines that interacts directly with internal structures of classes to obtain members and other requested data. It was found that this introspection code uses several built-in integrity tests and internal checking that produced wrong behavior if we change the aforementioned pointer. These checks are also invoked in several parts of the system to maintain class integrity, so it is not possible to modify them.

The implementation of the reflective primitives in our previous work forced us to use an auxiliary *SSCLI* class in order to store the information we need (Redondo et al., 2008). This element is called the *SyncBlock*, and every object and class in the system have one attached to its internal structure. We use this *SyncBlock* to our advantage in order to implement the desired functionality and solve the aforementioned implementation problems.

By storing a pointer to the dynamically assigned base class in this *Syncblock*, we can modify the implementation of the desired introspection primitives without modifying the "real" base type pointer. This way, any introspection primitive implementation in the system (*Type.getMethods()*, *Type.getFields()*, *Object.GetType()*, ...) could be modified to produce correct results, taking into account the dynamically assigned base class only if it is present. By using this approach we achieve two main benefits:

1. If no new base type is assigned, the original code will be executed, causing no performance loss.
2. The built-in integrity checks will continue working, since the static relationships between classes established at compile time appear to be left unchanged.

5 CONCLUSIONS

The major contribution of our work is the design and implementation of language-neutral dynamic inheritance support over a production high-performance JIT-based virtual machine. Nowadays, there is an active research line whose aim is to give more support to dynamic languages using JIT-based virtual machines (OpenJDK, 2008) (Chiles, 2008). Our system can be classified into this research line, trying a different approach to attain these objectives. Our dynamic inheritance support is combined with the already implemented reflective primitives, to offer complete support to structural reflection fully integrated into the machine internals. This enables the machine to offer the following benefits:

1. Any language (class or prototype-based) could benefit from structural reflection without altering its specifications. Semantics of this primitive is defined for both object models.
2. Low-level support of all the structural reflection primitives into the virtual machine, allowing any dynamic language to be completely implemented over its services, without adding any extra abstraction layer to simulate dynamic features.
3. Extend the interoperability present in the original machine to include dynamic languages, enabling them to interoperate with static ones.
4. Full backward compatibility with legacy code, since the intermediate code of the machine is not syntactically changed.
5. Due to the commercial character of the original virtual machine, it is possible to directly offer its new services to existing frameworks and languages designed to work with it.

Future work will incorporate meta-classes to the machine, taking advantage of the existing dynamic features. We are also developing a language that supports both dynamic and static typing, making the most of our ЯROTOR implementation (Ortin, 2008).

ACKNOWLEDGEMENTS

This work is funded by *Microsoft Research* with the project entitled *Extending dynamic features of the SSCLI*. It is part of a project who also define a programming language capable of offering both static and dynamic typing (Ortin, 2008).

Our work is supported by the *Computational Reflection research group* (`http://www.reflection.uniovi.es`) of the University of Oviedo (Spain).

The materials presented in this paper are available in `http://www.reflection.uniovi.es/rrotor`.

REFERENCES

Borning, A. H. (1986). Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40.

Bracha, G. and Griswold, D. (1993). Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA 93, ACM SIGPLAN Notices, volume 28*, pages 215–230.

Cazzola, W., Chiba, S., and Saake, G. (2004). Evolvable pattern implementations need generic aspects. In *ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, pages 111–126.

Chambers, C. and Ungar, D. (1989). Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM PLDI Conference*.

Chiles, B. (2008). CLR inside out: IronPython and the Dynamic Lang. Runtime. http://msdn2.microsoft.com/en-us/magazine/cc163344.aspx.

CodeHaus (2008). Boo. a wrist friendly language for the CLI. http://boo.codehaus.org/.

Cutsem, T. V., Mostinckx, S., Boix, E. G., Dedecker, J., and Meuter, W. D. (2007). AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society, SCCC 2007*.

Diehl, S., Hartel, P., and Sestoft, P. (2000). Abstract machines for programming language implementation. In *Future Generation Computer Systems*, page 739.

Ernst, E. (1999). Dynamic inheritance in a statically typed language. *Nordic Journal of Computing*, 6(1):72–92.

Golm, M. and Kleinder, J. (1997). MetaJava - a platform for adaptable operating system mechanisms. In *LNCS 1357*, page 507.

Grogan, M. (2008). JSR 223. scripting for the Java platform. http://www.jcp.org/en/jsr/detail?id=223.

Kleinder, J. and Golm, G. (1996). MetaJava: An efficient run-time meta architecture for Java. In *International Workshop on Object Orientation in Operating Systems*, pages 420–427.

Lucas, C., Mens, K., and Steyaert, P. (1995). Typing dynamic inheritance: A trade-off between substitutability and extensibility. Technical Report vub-prog-tr-95-03, Vrije Universiteit Brussel.

Maes, P. (1987). *Computational Reflection*. PhD thesis, Vrije Universiteit.

Meijer, E. and Drayton, P. (2004). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages*.

Meijer, E. and Gough, J. (2000). Technical overview of the CLR. Technical report, Microsoft.

MetaSlash (2008). PyChecker: a Python source code checking tool. http://pychecker.sourceforge.net/.

Nierstrasz, O., Bergel, A., Denker, M., Ducasse, S., Gaelli, M., and Wuyts, R. (2005). On the revival of dynamic languages. In *Software Composition 2005, LNCS*.

OpenJDK (2008). The Da Vinci machine. http://openjdk.java.net/projects/mlvm/.

Ortin, F. (2008). The StaDyn programming language. http://www.reflection.uniovi.es/stadyn/.

Ortin, F. and Cueva, J. M. (2004). Dynamic adaptation of application aspects. In *Journal of Systems and Software*. Elsevier.

Ortin, F., Lopez, B., and Perez-Schofield, J. B. (2004). Separating adaptable persistence attributes through computational reflection. In *IEEE Soft., Vol. 21, Issue 6*.

Ortin, F., Redondo, J. M., Vinuesa, L., and Cueva, J. M. (2005). Adding structural reflection to the SSCLI. In *Journal of .Net Technologies*, pages 151–162.

Pierce, B. P. (2002). *Types and Programming Languages*. The MIT Press.

Project, T. (2008). The Tunes project. http://slate.tunes.org/.

Redondo, J. M., Ortin, F., and Cueva, J. M. (2006a). Diseño de primitivas de reflexión estructural eficientes integradas en SSCLI. In *Proceedings of the JISBD 06*.

Redondo, J. M., Ortin, F., and Cueva, J. M. (2006b). Optimización de las primitivas de reflexión ofrecidas por los lenguajes dinámicos. In *Proceedings of the PROLE 06*, pages 53–64.

Redondo, J. M., Ortin, F., and Cueva, J. M. (2008). Optimizing reflective primitives of dynamic languages. In *Int. Journal of Soft. Engineering and Knowledge Engineering*. World Scientific.

Rose, J. (2008). JSR 292. supporting dynamically typed languages on the Java platform. http://www.jcp.org/en/jsr/detail?id=292.

Rossum, G. V. and Drake, F. L. (2003). *The Python Language Reference Manual*. Network Theory.

Singer, J. (2003). JVM versus CLR: a comparative study. In *ACM Proceedings of the 2nd international conference on principles and practice of programming in Java*.

Stutz, D., Neward, T., and Shilling, G. (2003). *Shared Source CLI Essentials*. O'Reilly.

Taivalsaari, A. (1992). Kevo: A prototype-based OO language based on concatenation and module operations. Technical report, U. of Victoria, British Columbia.

Thomas, D., Fowler, C., and Hunt, A. (2004). *Programming Ruby*. Addison-Wesley Professional, 2nd edition.

Thomas, D., Hansson, D. H., Schwarz, A., Fuchs, T., Breed, L., and Clark, M. (2005). *Agile Web Development with Rails. A Pragmatic Guide*. Pragmatic Bookshelf.

Udell, J. (2003). D. languages and v. machines. Infoworld.

Ungar, D., Chambers, G., Chang, B. W., and Holzl, U. (1991). Organizing programs without classes. In *Lisp and Symbolic Computation*.

Wolczko, M., Agesen, O., and Ungar, D. (1996). Towards a universal implementation substrate for object-oriented languages. Sun Microsystems Laboratories.