Supporting Efficient Dynamic Aspects Through Reflection and Dynamic Compilation

Frédéric Ogel
France Telecom R&D
38-40, rue du général Leclerc
92120 Issy Les Moulineaux, France
frederic.ogel@inria.fr

Gaël Thomas
LIP6, Université Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France
gael.thomas@lip6.fr

Bertil Folliot
LIP6, Université Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France
bertil.folliot@lip6.fr

ABSTRACT

As systems grow more and more complex, raising severe evolution and management difficulties, computationnal reflection and aspect-orientation have proven to enforce separation of concerns principles and thus to address those issues. However, most of the existing solutions rely either on a static source code manipulation or on the introduction of extra-code (and overhead) to support dynamic adaptation. Whereas those approaches represent the extreme of a spectre, developpers are left with this rigid tradeoff between performance and dynamism. A first step toward a solution was the introduction of specialized virtual machines to support dynamic aspects into the core of the execution engine. However, using such dedicated runtimes limits applications' portability and interoperability.

In order to reconcile dynamism and performance without introducing portability and interoperability issues, we propose a dynamic reflexive runtime that uses reflection and dynamic compilation to allow application-specific dynamic weaving strategies, whithout introducing extra-overhead compared to static monolithic weavers.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]:

General Terms

Design

Keywords

Computational Reflection, AOP, Performance

1. INTRODUCTION

Nowadays systems are growing more and more complex, thus increasing development costs and raising severe evolution and maintenance difficulties. Most often, integrating an unanticipated feature into a traditional monolithic application requires an in-depth knowledge of internal interactions and side-effects and is both time-consuming and error-prone. In response to those issues, *Separation of Concern* principles have been proposed and techniques such as computational reflection, aspect-orientation and subject-orientation have emerged as a solution toward these principles.

However, most of those solutions are still relying on *static* manipulation of applications' source code or intermediate representation,[1] hence preventing any *live* adaptation in response to unexpected changes. Higher level of dynamism have been proposed through the introduction of *interceptors*, which results in extra-overheads. Developers are thus left with a rigid tradeoff between dynamism and performance. On one hand, static solutions, as AspectJ [12], rely on a compiler for code specialization and aspects wrapping. The final application is thus as rigid as any standard application. On the other hand, dynamic approaches support late bindings. Hence, they allow more parameters, such as available resources or execution conditions, to be taken into account, and potentially enable support for dealing with unexpected changes. Such dynamic solutions, as Jac [1] for aspect-oriented programming (AOP) in Java, rely on the introduction of indirections and thus come at the price of some performance overhead. Moreover, the weaver is still a rigid and closed component: typically the weaving strategy can not be adapted to application-specific semantics and constraints. The main reason is because this tradeoff is fixed during the weaver conception.

A first step toward reconciling performance and dynamism consists in runtime-level aspects weaving, as with dedicated virtual machines. Such specialized virtual machines provide support for aspects weaving at a lower level, typically in the core of the execution engine, hence eliminating the need for meta-level indirections and resulting in far better performance. However, such solutions lead to a proliferation of ad-hoc virtual machines poorly interoperable, thus seriously limiting the portability.

Based on those observations, we have proposed a dynamic reflexive runtime, called the MICRO-VM, which uses reflection and dynamic compilation to provide support for efficient dynamic aspects at the lowest possible level, hence reconciling dynamism and performance. Moreover, the MICRO-VM defines a common language substrate upon which ap-

[1]such as Java bytecodes.

plications build a flexible execution environments dedicated to themselves, avoiding the interoperability and portability issues raised by specialized virtual machines and traditional *one-size-fits-all* approaches. This runtime is based on a language and hardware independent platform called the Virtual Virtual Machine (VVM) [15].

The remainder of this paper starts by presenting main techniques for supporting aspects and reflection, focusing on Java environments since they are representative of the work done in this area. Section 3 gives an overview of the MICRO-VM's architecture, while Section 4 describes the mechanisms supporting dynamic aspects. An application to the construction of an aspect-enabled Java virtual machine is presented in Section 5, followed by some conclusions and perspectives in Section 6.

2. RELATED WORK

Solutions for introducing reflection or aspect-orientation in Java environments fall into three categories: (i) using compilation techniques, (ii) using bytecodes modification, (iii) using a specialized virtual machine. However, each of these approaches rely on interception of both method calls and field manipulations. In the remaining of this section, we use the term *interception points* to denote both join points and reflection points.

Compilation-based techniques consist in statically inserting *interception points* during the compilation of the application. Reflective Java [27] and AspectJ [12] use an external language to describe the transformation of the application's source code into an augmented version containing calls to the meta-objects (for Reflective Java) or Code Advice (AspectJ). OpenJava [21, 22] extendsthe Java language with new keywords dedicated to *interception points* definition. As interception code is inserted before compilation, no indirection to the Java reflection API is needed, resulting in optimized applications' code. However, such solutions allow *a-priori* insertion of interception points, hence a lack of dynamism (the number of interception points is statically fixed) and a lack of support for introspection of interception points.

Solutions based on bytecodes rewriting allow the deployment of aspects and reflexive protocols into compiled code. Not only source code is no longer required for modification, but integration of interception points can be postponed (delayed) to execution time.

Compilation-based technique and bytecode rewriting offer the same possibility, but the adaptation don't need to access to source code with bytecode rewriting.Since it uses the Java reflection API to redirect interception points to target methods performances are still poor. Tools such as BCel [9] and Javassist [7, 8] allows to modify existing bytecodes at load-time. Dalang [24] is oriented toward reflection and thus encapsulate both functional code and meta-level code in an automatically generated class.

Kava [26, 25] allows to encapsulate sequences of bytecodes and to redirect them to meta-objects. The three main aspect oriented platforms, Jac [14, 1], JBoss OAP [2] and AspectWerkz [6], are using bytecodes rewriting to inert interception points into Java bytecodes. Advice are expressed using the Java language (Jac), XML-based configuration files (JBoss and AspectWerkz) or comments in the Java source code (AspectWerkz).

Using a specialized virtual machine allows to deal directly with internal representation of methods, exceptions and fields and thus to optimize interception points. In addition, weaving and meta-objects protocols can be defined inside the virtual machine. MetaXa [10, 11] proposes a layered MOP, Guaraná [16, 19, 18, 17] allows to (re)define the sequence of invocations on meta-objects associated with a base object. SteamLoom [5] uses a modified Java virtual machine (based on IBM's JikesRVM [3]) to weave aspects at the virtual machine level, resulting in far better performance.

Specialized virtual machine based approaches are thus both more dynamic and more efficient solutions: the set of interception points can be dynamically extended while the modified virtual machine avoids indirections to the Java reflection API. However, using specialized virtual machines limits the portability of applications specifically developed for such platforms. Thus, we investigated a solution to let the application decides what kind of specialized virtual machine match best its needs. Our approach offers both the same level of portability as compilation-based technique or bytecode rewriting, and the same dynamism and performance as dedicated virtual machine.

3. THE MICRO-VM RUNTIME

Our approach relies on a reflective and minimal execution environment, called the MICRO-VM.[2]

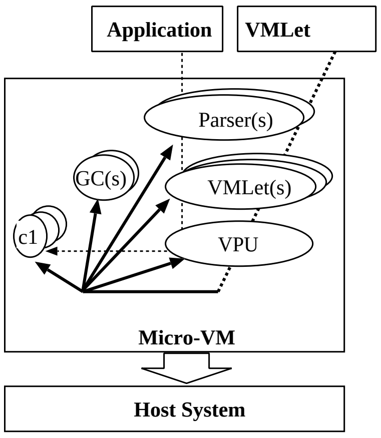

Figure 1: An overview of the Micro-VM.

The MICRO-VM is dedicated to building execution environments. It is structured as a set of interfaces and components, which are based on the ODP Reference Model [13]. The MICRO-VM is both a minimal virtual machine (since it can execute abstract instructions) and a dynamic compiler. It offers full flexibility because of two properties: (i) the MICRO-VM is entirely open and reflexive, thus anything (from its internals to application code) can be adapted; (ii) the dynamic compilation inherent in the MICRO-VM allows *on-the-fly* reconfiguration.

As illustrated in Figure 1, applications are loaded[3], parsed and compiled into a native representation of their compo-

[2]For a more detailed description of the MICRO-VM, previously called YNVM, see [15].

[3]Application loading can rely on the default MICRO-VM loader or any format-specific loader previously loaded into the MICRO-VM.

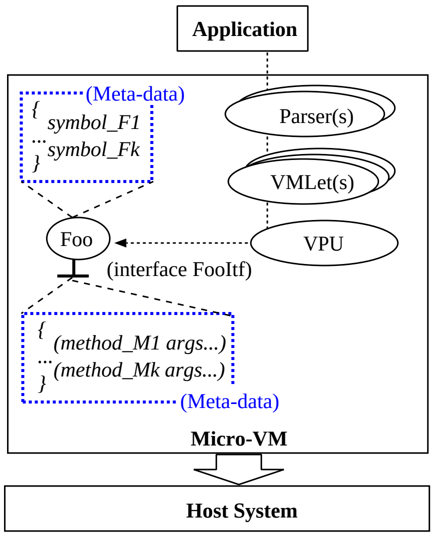

Figure 2: Reification of a component

nents by the MICRO-VM's dynamic compilation chain. The compilation is done through a Virtual Processor Unit, called the VPU [20]. The VPU is an abstract stack machine which tranfrom a sequence of abstract instruction into an asembly code. VMLets are used to manipulate and modify both the MICRO-VM internals and applications.

This minimal execution environment is then extended or specialized according to a given application domain (as with a *Domain Specific Language*-based approach). This adaptation consists of loading a VMLet, which describes a dedicated execution environment. This specification directly extends the MICRO-VM with new dedicated primitives, operators, abstractions for resource management, or language support for dynamic compilation. While remaining a generic virtual machine dedicated to execution environment construction, the MICRO-VM however becomes a native, domain-specific execution environment too, offering the same semantic and performance as any other native, hand-coded, domain-specific execution environment.

Applications executed by the MICRO-VM are called *active applications*. They are composed of two distinct parts: the application code (for example, a JAVA class file) and additionnal VMLets loaded by the MICRO-VM. Those VMLets adapt the MICRO-VM to the application's needs. For example, an application based on a JVM with persistent objects is composed of a JAVA program (the application), a Java VMLet defining a standard JVM and an additionnal VMLet adapting Java objects' storage.

4. DYNAMIC ASPECTS SUPPORT IN THE MICRO-VM

As a reflexive runtime, the MICRO-VM provides a meta-level. Meta-data are kept from the dynamic compilation and associated with components, symbols, interfaces and primitive objects (expressions, internal objects, syntaxes and functions). Figure 2 illustrates the relation between data[4] and meta-data. An application has defined a component, namely Foo, which is compiled from its high-level description into a native form through the dynamic compilation chain. This dynamic compilation chain includes parsing, as the MICRO-VM supports different input languages but rely on an AST-based abstract code representation,[5] domain-specific code-transformation/analysis (through the associated VMLets) and compilation of the intermediate representation into native code. Meta-data associated with the resulting component includes the name of the interface it implements. This interface (FooItf in the figure), which is a *first-class* object, has associated meta-data resulting from its compilation, including its abstract representation (a list of methods' signatures in the figure). The list of functions' symbols that a component uses to implement an interface is also kept and associated to its meta-data. In turn, each of those symbols has the abstract representation of the corresponding function associated to its meta-data. Therefore, given a component's symbol, the MICRO-VM can retrieve its methods, the functions which implement them and access their abstract code representation. Such intermediate representations are manipulated as simple lists of expressions (AST) to generate modified versions, through dynamic compilation of the new abstract code representations.

```
   (define .FooItf                        ;; the meta-description
     (def-interface "simple-interface" ;; its name
       .meth))                            ;; and one method

 5 (define .Foo                           ;; the meta-component
     (implement "simple-component" ;; its name
                FooItf                    ;; implement the FooItf interface
                (lambda(comp buf) ;; called with a println
                  (:component.write-str buf "SimpleInterface<")
10                (:component._print (name comp) buf)
                  (:component.write-str buf ">"))
                0                         ;; the finalizer
                [(lambda(comp x)
                   (:system.printf "; [%d], call meth on \"%s\"\n"
15                                 x
                                   (:object.print-string comp)))
                 ]
                ;; the fields of this component
                [word name]))
```

Figure 3: Construction of a component.

The figure 3 illustrates how components are defined. The FooItf interface is defined at line 1 with one method called meth. A component Foo is defined at line 5: it implements the FooItf interface and have a field called name (at line 19). With the definition of FooItf, a meta-interface is allocated[6]. Within this meta-interface, a meth symbol is defined. This symbol is a macro which allows an application to call the method on a component. This call is inlined in the caller. The interface of a component is a virtual table. Each entry contains the implementation of a method of the interface.

[4]This includes executable code, which we consider code to be a special form of data.

[5]The MICRO-VM uses a common Lisp-like Abstract Syntax Tree (AST) intermediate representation.

[6]Such meta-interface are collected by the underlying garbage collector.

Symbols generated during the definition of interfaces are also used to store methods' offsets in this virtual table. For example, the meth symbol store the offset of the method in the virtual table. Methods' offests are then used to modify components.

```
   (define .ast (:object.method.definition ;;; the ast of
                  (:object.method.deref   ;;; the method associated with
                   (word                  ;;; the code at
                    (+ meth               ;;; "offset of meth" in
5                      (:component.md._vt Foo)))))) ;; Foo's virtual table

   (set! (word (+ (:component.md._vt simple−component) meth))
         ;; replace the code with
         (lambda(input x)
10         (:system.printf "A new method\n")
           ;; we can put the old ast here
           ))
```

Figure 4: On the fly modification of a component.

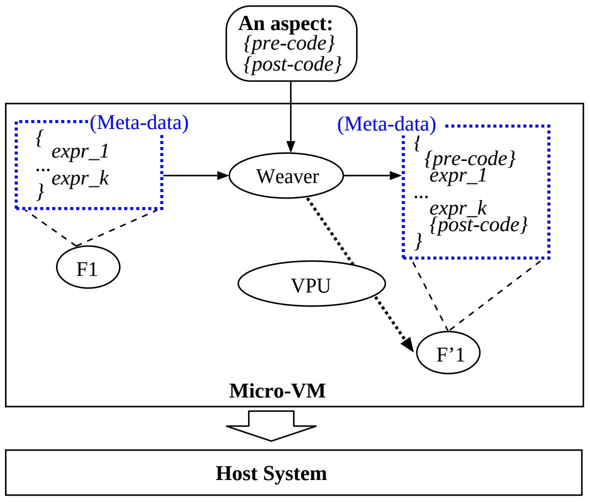

Figure 5: Dynamic generation of an aspectified method

The figure 4 gives an example of a component's dynamic modification. At line 1, the previous AST is stored in an ast symbol. Then, this AST is modified and stored in the component's virtual table. The reflexive layer of the MICRO-VM does not change the execution time (the aspects are inlined in the method), but a lot of meta-data are preserved (the cost in memory can be important). Using virtual tables has the same cost as the virtual calls in C++.

The MICRO-VM is written in C and the internal components use the same format (virtual tables and fields). Meta-data are generated through the compilation chain of the MICRO-VM and then loaded on demand: a VMLet can modify the internal components of the MICRO-VM, in particulary the entry syntax[7].

Let us consider a generic example of dynamic weaving which uses this meta-level: a Weaver component is given an aspect to wrap around the M1 method of a Foo component. As illustrated in Figure 2, the weaver uses the meta-data associated with the Foo component to get the symbol associated with the function implementing the M1 method for the component (F1 in the Figure). The meta-data associated with the F1 symbol contains the intermediate representation describing this function's code (as a list of expressions). As represented in Figure 5, given the abstract representation defining the aspect (both pre-code and post-code) and the abstract code representation of the target F1 function, the weaver produces a new abstract representation (a simple list object) containing both the aspect's code and the original set of expressions corresponding to the target function. Then, this new AST is dynamically compiled into native code through a call to the underlying VPU and the weaver replace the original reference to the F1 symbol by a reference to the freshly generated function (F'1 in the figure). Hence aspects are wrapped *on-the-fly* without the traditional overhead tied to interception techniques.

The performance of such dynamic weaving is heavily tied to those of the dynamic compilation chain. For example, weaving a simple aspect (a dozen of high-level instructions) around a bigger method cost approximately a hundred microseconds on a 290MHz G3 PowerPC running Linux 2.4. This corresponds to the insertion of the aspect's code into the abstract representation of the method and the dynamic generation of a new native representation.

[7]We can not generate AST for the internal functions, but we plan to (re-)write the MICRO-VM in MICRO-VM to generate a new binary from the assembly code.

5. APPLICATION TO A JAVA VIRTUAL MACHINE

In order to illustrate our approach, we present an aspect-enabled Java virtual machine, which provides dynamic aspects support into the core of the execution engine. As opposed to specialized runtime approaches, such as SteamLoom, which provides (and requires) dedicated virtual machines, hence limiting portability and interoperability, our solution rely on the dynamic adaptation of the Java runtime. We have developed a new VMLet: a complete and standard Java virtual machine built with the component inherited from the MICRO-VM [23]. The aspect weaver described in the previous section is used to modify the behavior of the Java virtual machine dynamically. We used it to build a Java applicative weaver inspired from SteamLoom [5]. The active application is splited in two part : a standard Java binary and two VMLets. The first one is the Java virtual machine and the second the applicative aspect specialisation. The second one can be loaded during the execution of the application. Our Java virtual machine support for dynamic aspect weaving allows:

- a high-level of dynamism and flexibility (Java aspects are weaved/un-weaved *on-the-fly*);
- good performance (no indirection to the Java reflection API, dynamically inlined aspect's code);
- application-specific aspect model and strategies (aspects support is dynamically integrated into the runtime, potentially by the application itself);
- a high level of portability (the active application spe-

cialize the Java VMLet and create the applicative aspect weaver).

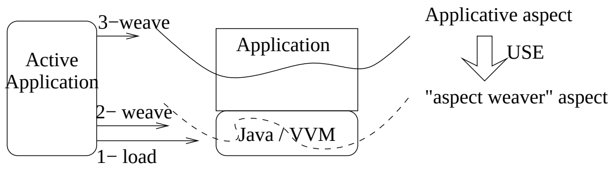

Figure 6: Two levels of aspects

Figure 6 represents those two levels of aspects weaving. The active application loads the standard Java VMLet into the MICRO-VM, then it weaves the low-level aspect weaver (the *weaving* aspect) in the resulting Java virtual machine, and finally uses this internal weaver to weave application-level aspects. Such an approach eliminates the portability and interoperability issues raised with statically specialized virtual machines. Moreover, it allows aspects-related strategies to be application-specific.

The *weaving* aspect uses directly the virtual machine internal functions to retrieve the description of the methods around which to weave aspects. The MICRO-VM's language is extended with new aspect-related primitives, such as (:aspect.weave-on-meth <a-method> <a-pointcut>) for assigning the address of <a-pointcut> to <a-method>.

Figure 7 illustrates how an aspect is defined and weaved on a pointcut. The desc symbol holds a pointer to the internal abstract code representation of the associated method and the orig symbol is a copy of the original method (hence the name). The fct() function is weaved on desc and simply outputs the name of the method which is going to be invoked. The (:aspect.invoke-virtual orig value obj) expression invokes the original method, with an *inlined* call to fct(), as any standard Java method call: we benefit from the dynamic compiler (MICRO-VM's VPU) to speed-up the invocation of weaved methods.

```
(define .class (get−class (jvm) "Hello"))
(define .desc (get−method class "m" "(I)V"
                          ACC_VIRTUAL))
(define .orig (dup−method desc "m-orig"))
(define .fct
  (lambda(blackbox value this)
    (let ([meth (blackbox.method cache)])
      (printf "; call: %s\n" (print−string meth))
      (invoke−virtual orig value this)
      (printf "; end of: %s\n" (print−string meth)))))

(wave−on−method m fct)
```

Figure 7: An example of application-level aspect weaving.

Whereas the fct() function typically calls the original method, it can call any other methods from any other classes. We did not define a proceed keyword as in Jac, but it is semantically equivalent. As methods' abstract representations are augmented with aspects abstract representations (through list manipulations), aspects can be un-weaved. The resulting code is equivalent to the one statically produced by any static aspect-weaver (as AspectJ), while offering a high level of dynamism.

Another example of dynamic specialization of a Java virtual machine for aspects support consists in dynamically weave aspects related to escape-analysis [4]. Applications are processed by analytical tools and extra-memory management information are embedded within their bytecode representations. Thus, dedicated virtual machines are required in order to take advantage of those extra-informations. An example of extending a MICRO-VM-based Java virtual machine with escape-analysis support using aspects dynamically weaved into the virtual machine's core is given in [23].

6. CONCLUSION AND PERSPECTIVES

Based on a reflexive dynamically extensible runtime we proposed a solution to reconcile dynamic aspects and performance, without introducing any portability issues. By using dynamic specialization of applications' runtime, aspect-related strategies are application-specific. Thanks to the efficient underlying dynamic compiler, dynamic weaving performance seems to be promising. Moreover, once weaved, aspects do not introduce any indirections, hence overheads, to applications' code.

Even if the underlying reflexive MICRO-VM provides mechanisms for security/integrity checks on aspects' interactions, our *weaving* aspect does not perform such verifications.

We plan to investigate further interoperability issues and especially low-level aspects support in a multi-language environment. For example, weaving C++ aspects in Java applications.

7. REFERENCES

[1] Jac homepage. http://jac.aopsys.com/.

[2] Jboss aop home page. http://www.jboss.org/products/aop.

[3] The jikes research virtual machine. http://www-124.ibm.com/developerworks/oss/jikesrvm/.

[4] B. Blanchet. Escape Analysis for Java : Theory and Practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd AOSD Conference*, pages 83–92, Lancaster, UK, march 2004.

[6] J. Boner. Aspectwerkz – dynamic aop for java. http://codehaus.org/ jboner/papers/aosd2004_aspectwerkz.pdf, 2004.

[7] S. Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct. 1998.

[8] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the second international conference on Generative programming and component engineering table (GPCE'03)*, pages 364 – 376, Erfurt, Germany, 2003.

[9] M. Dahn, J. van Zyl, and E. Haase. *BCEL - The ByteCode Engineneering Library - Manual.* http://jakarta.apache.org/bcel/manual.html.

[10] M. Golm. Design and implementation of a meta architecture for java. Master's thesis, University of Erlangen, Jan 1997.

[11] M. Golm. metaxa and the future of reflection. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, Oct. 1998.

[12] J. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* 2003.

[13] ISO International Organization for Standardization. *Information Technology – Open Distributed Processing – Reference Model*, ISO/IEC 10746-(1-4) edition, 1996-1998.

[14] ObjectWeb. *Aspect-Oriented Programming with JAC.* http://jac.objectweb.org/docs/programmer_guide.html.

[15] F. Ogel, G. Thomas, I. Piumarta, A. Galland, B. Folliot, and C. Baillarguet. Towards Active Applications: the Virtual Virtual Machine Approach. *New Trends in Computer Science and Engineering*, 2003.

[16] A. Oliva and L. Buzato. The design and implementation of guaraná. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, San Diego, California, USA, May 1999.

[17] A. Oliva, I. Garcia, and L. Buzato. Guaraná: a tutorial. Technical report ic-98-31, Univerisidade Estadual de Campinas, Sept. 1998.

[18] A. Oliva, I. Garcia, and L. Buzato. The implementation of guaraná on java. Technical report ic-98-32, Univerisidade Estadual de Campinas, Sept. 1998.

[19] A. Oliva, I. Garcia, and L. Buzato. The reflective architecture of guaraná. Technical report ic-98-14, Univerisidade Estadual de Campinas, Sept. 1998.

[20] I. Piumarta. The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation. In *the 3rd USENIX symposium on Virtual Machine Research and Technology*, San Jose, California, USA, may 2004.

[21] M. Tatsubori. An extension mechanism for the java language. Master's thesis, Tokyo Institute of Technology, University of Tsukuba, 1999. Master of Engineering Dissertation, Graduate School of Engineering.

[22] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *Reflection and Software Ingineering*, pages 119–135. Lecture Notes in Computer Science 1826, Springer-Verlag, July 2000.

[23] G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a flexible Java runtime upon a flexible compiler. *International Journal on Computers and Applications Special Issue on 'System & Networking for Smart Objects'*, january 2005.

[24] I. Welch and R. Stroud. Dalang — a reflective extension for java. Technical Report CS-TR-672, University of Newcastle upon Tyne, September 1999.

[25] I. Welch and R. Stroud. Kava - using bytecode rewriting to add behavioural reflection to java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, USA, January 2001. USENIX Association 2001.

[26] I. Welch and R. J. Stroud. From dalang to kava — the evolution of a reflective java extension. In *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, Saint-Malo, France, July 1999.

[27] Z. Wu and S. Schwiderski. Reflextive java: Making java even more reflexive. Ansa phase 3, ANSA – APM Limited, Cambridge, UK, Feb 1997.