MetaJava — A Platform for Adaptable Operating-System Mechanisms

Jürgen Kleinöder, Michael Golm

August 1997 TR-I4-97-12

Technical Report

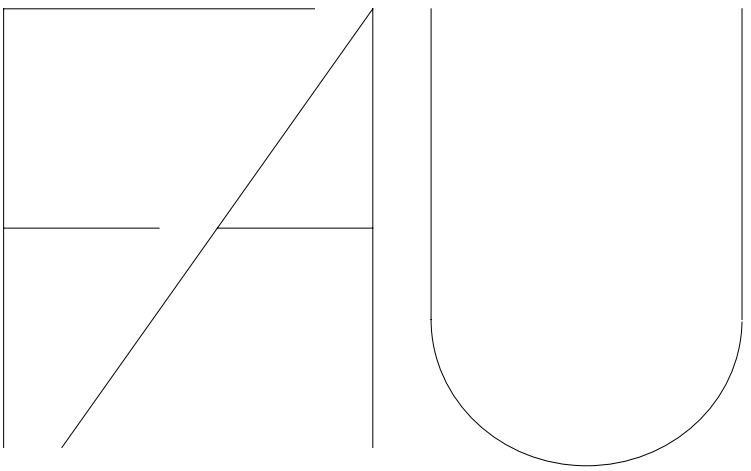
Department of Computer Science

Operating Systems (IMMD IV)

Univ. of Erlangen • IMMD IV
Martensstr. 1
D-91058 Erlangen
Germany

Phone: +49.9131.85.72 77
Fax: +49.9131.85.87 32
URL: http://www4.informatik.uni-erlangen.de

Friedrich-Alexander-Universität Erlangen-Nürnberg Germany



This paper was published as:

Jürgen Kleinöder, Michael Golm: MetaJava - A Platform for Adaptable Operating-System Mechanisms. *ECOOP '97 - Workshops*; Lecture Notes in Computer Science; Springer, Berlin, Heidelberg, to appear 1997.

MetaJava - A Platform for Adaptable Operating-System Mechanisms

Jürgen Kleinöder, Michael Golm

University of Erlangen-Nürnberg, Dept. of Computer Science IV[1]
Martensstr. 1, D-91058 Erlangen, Germany
{kleinoeder, golm}@informatik.uni-erlangen.de

Abstract. Fine-grained adaptable operating-system services can not be implemented with today's layered operating-system architectures. Structuring a system in base level and meta level opens the implementation of operating-system services. Operating-system and run-time services are implemented by meta objects which can be replaced by other meta objects if the application requires this.

1 Introduction

This paper describes the MetaJava system, an extended Java interpreter that allows structural and behavioral reflection.

MetaJava was built to achieve the following goals:

- It must be possible to separate the functional, application-specific concerns from non-functional concerns, such as parallelism or fault tolerance.
- The reflective architecture must be general; different problems, such as persistence, distribution, replication, or synchronization must be solvable in it.

The paper is structured as follows. We first introduce the concepts of reflection and metaprogramming in Section 2 and the computational model of MetaJava in Section 3. Section 4 explains how object-oriented meta-level architectures can be applied to the structuring of operating-system mechanisms. Section 5 addresses the problems that had to be solved for the implementation of MetaJava. Section 6 discusses related work and Section 7 concludes the paper and suggests future work.

2 Reflection and Metaprogramming

In the past, programs had to fulfill a task in a limited computational domain. As applications have become increasingly complex today, they need more capable programming models, which support mechanisms of modern run-time environments, such as multithreading (synchronization, dead-lock detection, etc.), fault tolerance, distribution, mobile objects, extended transaction models, persistence, and so on. Many ad hoc extensions to languages and run-time systems have been

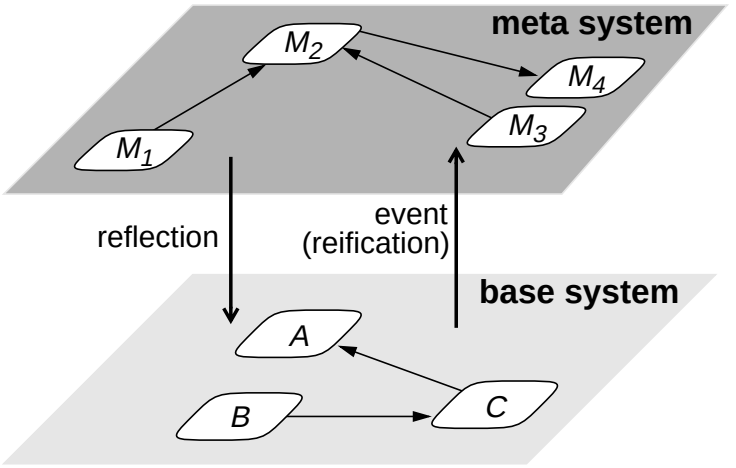
1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* Grant *Sonderforschungsbereich SFB 182*, Project *B2*.

implemented to support some of these mechanisms. But such proceeding does not meet the very diverse demands, different applications make on their run-time environment. Instead it is desirable to provide an application with means to control the implementation of its run-time mechanisms and to add own modifications or extensions if necessary. Reflection is a fundamental concept to get influence on properties and implementation of the execution environment of a computing system.

Smith's work on 3-LISP [20] was the first that considered reflection as essential part of the programming model. Maes [13] studied reflection in object oriented systems. According to Maes *reflection* is the capability of a computational system to "reason about and act upon itself" and adjust itself to changing conditions. *Metaprogramming* separates functional from non-functional code. *Functional code* is concerned with computations about the application's domain (*base level*), non-functional code resides at the *meta level*, supervising and managing the execution of the functional code. To enable this supervision, some aspects of the base-level computation must be reified at the meta level. *Reification* is the process of making something explicit that is normally not part of the language or programming model. The advantages of the separation in base level and meta level are outlined in [9].

As pointed out in [3] there are two types of reflection: structural and behavioral reflection (in [3] termed computational reflection). Structural reflection reifies structural aspects of a program, such as inheritance and data types. The Java Reflection API [21] is an example for structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. The main focus of MetaJava is to provide behavioral reflection capabilities.

3 Computational Model of MetaJava



The computations of objects A, B, and C raise events that transfer control to the meta level. The meta objects influence the computation of A, B, and C.

Fig. 1 Computational model of behavioral reflection

Traditional systems consist of an operating system (OS) and, on top of it, a program which uses the OS services through an application programmer interface (API). Additional services may be provided by a run-time layer between OS and application.

As OS and run-time services supervise and manage state and execution of applications, their functionality corresponds to that of a meta level as defined in the previous section.

In our reflective model the system consists of a meta system and the application program (the *base system*). The program may not be aware of the meta system.

enter-method(object, method, arguments)
method is being called at object with arguments

load-class(classname)
class classname is being used for the first time and must be loaded

create-object(class)
an instance of class is being created

acquire-object-lock (object)
the lock of object is being acquired

release-object-lock(object)
the lock of object is being released

read-field (object, field)
the field of object is being read

write-field (object,field, value)
value is being written into the field of object

Fig. 2 Events generated by a base-level computation

The computation in the base system raises events (see Fig. 1). These events are delivered to the meta system. The meta system evaluates the events and reacts in a specific manner.All events are handled synchronously. The base-level computation is suspended while the meta object processes the event. This gives the meta level complete control over the activity in the base system. For instance, if the meta object receives a *enter-method* event, the default behavior would be to execute the method. But the meta object could also synchronize the method execution with the execution of another method of the base object. Other alternatives would be to queue the method for delayed execution and return to the caller immediately, or to execute the method on a different host. What actually happens depends entirely on the meta object used. The currently defined events are listed in Figure 2.

A base object also can invoke a method of the meta object directly. This is called explicit meta interaction and is used to control the meta level from the base level.

Not every object must have a meta object attached to it. Meta objects may be attached dynamically to base objects at run time. This is especially important if a distributed computation is controlled at the meta level and certain method arguments need to be made reflective. As long as no meta objects are attached to an application, our meta architecture does not cause any overhead. So applications only have to pay for the meta-system functionality where they really need it.

Currently meta objects can be attached to references, objects, and classes. If a meta object is attached to an object, the semantics of the object is changed. Sometimes it is desirable only to change the semantics of one reference to the object — for example, when tracing accesses to the reference, or when attaching a certain security policy to the reference [18]. Attaching a meta object to a class makes all instances of the class reflective.

To fulfill its tasks the meta object has access to a set of methods which can manipulate the internal state of the virtual machine. These methods are called the *meta-level interface* (MLI) of the virtual machine. Architecture and terminology of the Java VM are described in detail in [12]. A list of the most important methods of the MLI is given in Figure 3.

void ***attachObject*** (MetaObject meta, Object base)
Bind a meta object to a base object.

Object ***continueExecution*** (EventMethodCall event)
Continue the execution of a base-level method. This calls the non-reflective method. No event is generated, otherwise the reflection would not terminate.

Object ***doExecute*** (EventMethodCall event)
Execute a method. Contrary to the previous method, this one calls the method as if it were called by an ordinary base object.

Object ***createNewInstance*** (EventObjectCreation event)
Create a new instance of a class. The class name is passed as String (as part of the event parameter).

void ***installBytecode*** (Object ref, String method, byte code[])
Installs code as new method bytecode. Together with addConstantPoolItem this method is used to generate a stub method in the place of the original method.

String ***retrieveObjectLayout*** (Object ref)
Returns the types of all fields of object ref. This method is used together with getFieldObject, getFieldInt, getFieldFloat, setFieldObject, etc., to access fields of arbitrary objects.

Object ***getField*** (Object ref, String fieldName)
Returns the contents of field fieldName of object ref. Name and type of all fields (object layout) can be retrieved with retrieveObjectLayout.

void ***setField*** (Object ref, String fieldName, Object obj)
Sets the contents of field fieldName of object ref to object obj.

int ***addConstantPoolItem*** (Class c, CPItem i)
Adds an item to the constant pool of class c.

Fig. 3 Selected methods of the meta-level interface of the MetaJava virtual machine

4 Metaobjects for open operating-system implementations

In a Java environment, most mechanisms and policies that are traditionally regarded as operating-system or run-time–system services are provided either by the virtual machine itself or by native libraries. As these services are implemented in C and the flexibility and comfort of a Java environment is not available for them, it is not easy to adapt them to special application needs or to transparently add new services. MetaJava provides an architecture for an open implementation of most mechanisms and policies that are currently a fixed part of the virtual machine, such as memory management, garbage collection, thread management and scheduling, or class management. The virtual machine has to provide merely a very primitive implementation of a few basic mechanisms, such as thread switching, and simple policies to get the first metaobjects up and running (e.g. a simple class loader to install classes from a local disk). More complex mechanisms and policies can be implemented as Java metaobjects, which can use the basic mechanisms via the meta-level interface.

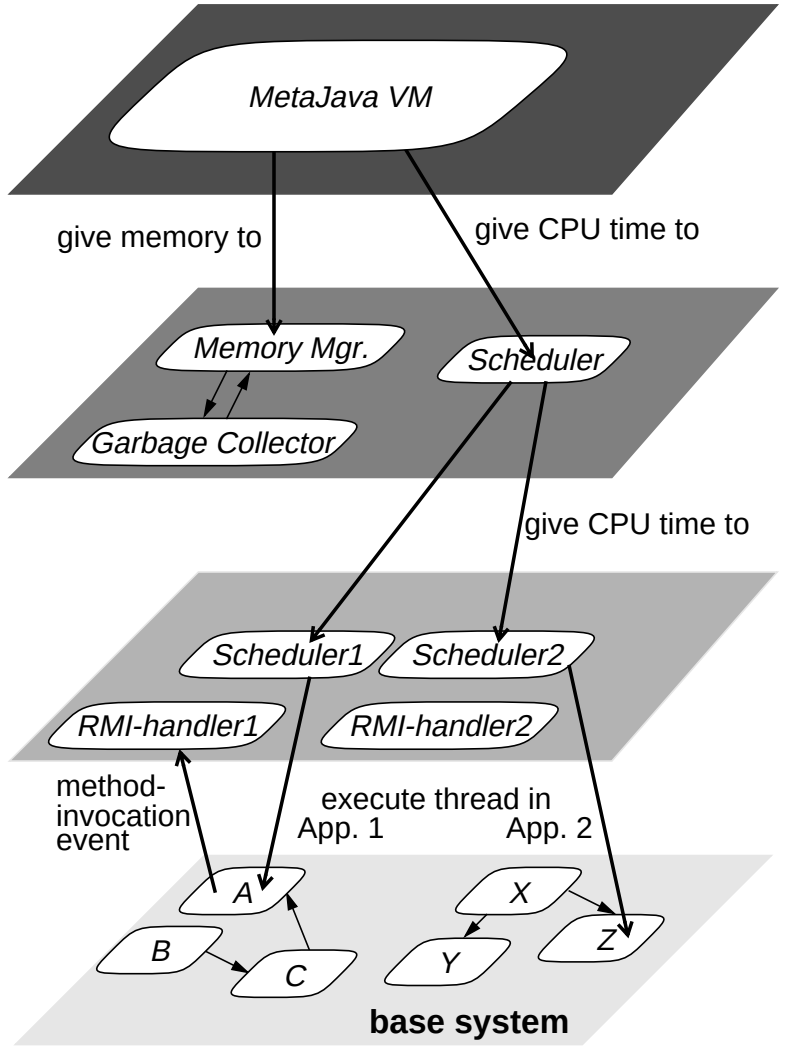


Fig. 4 Hierarchy of meta levels

If several applications are executing within one Java machine, application-specific metaobjects lead to a hierarchy of metaobjects (Fig. 4). Global metaobjects give the resources to applications, application-specific metaobjects control the resource usage within the application. Of course, even application parts may employ their own metaobjects, if necessary.

In addition to the mechanisms listed above, a broad range of extended run-time services can be implemented by metaobjects: persistence, object migration, object replication [10], just-in-time compilation, active objects [5], asynchronous method invocations, transactions, synchronization, various security policies, and so on.

5 Implementation

Integration into Java. The initial version of MetaJava used a shared library to extend Sun's Java Virtual Machine (JVM). The further development required extensive changes to the JVM, so we decided to build our own virtual machine, the MetaJava Virtual Machine (MJVM). The MJVM is a superset of the JVM. It uses the same class-file format and executes the same bytecode set as the JVM, but provides a meta-level interface to the virtual machine (Fig. 3).

We did not change the Java language or the class-file format, so off-the-shelf development tools can be used.

This section outlines the changes to the JVM that were necessary to enable our reflective model. A more detailed description can be found in [11].

5.1 Shadow classes

The purpose of a metaobject is to change the semantics of one object. This change should not affect other objects of the same class. Ferber [3] suggests to use classes for structural reflection and metaobjects for behavioral reflection. We do not adopt this model because we think that classes (conceptually) should contain complete information about the object. The separation as proposed in [3] seems only be justified by the fact that classes are not related to individual objects but to all instance of the class. Hence, the behavior of individual objects can not be changed by customizing classes.

We introduce shadow classes to solve this problem, which is inherent in class-based languages. A shadow class C' is an exact copy of class C with the following properties:

- C and C' are undistinguishable at the base level
- C' is identical to C except for modifications done by a meta-level program
- Static fields and methods are shared between C and C'.

The base-level system can not differentiate between a class and its shadow classes, so the shadowing is transparent to the base system. Several problems concerning class data consistency, class identity, garbage collection, code consistency, and memory consumption had to be tackled to ensure this transparency.

5.2 Event Mechanism

Shadow classes allow us, to transparently modify the class structure of individual objects. This is the base mechanism for the reification of object behavior.

- *Reification of incoming method invocations:* The method bytecodes are replaced by a stub code, which jumps to the meta space.
- *Reification of outgoing method calls:* The corresponding *invokevirtual* opcodes are replaced by stub code.
- *Reification of instance variable accesses:* The corresponding *putfield* and *getfield* opcodes are replaced by stub code.
- *Reification of object locking:* The MJVM uses a function pointer in the class structure to acquire or release an object lock. This pointer normally refers to a function that implements locking and unlocking. When a metaobject registers for the lock events, the MJVM redirects this pointer to the metaobject's event handler.
- *Reification of class loading and object creation:* Class loading and object creation is reified similar to object locking. The MJVM class structure contains a pointer to the class-loader/object-creation function. If a metaobject registers for this event, the pointer is redirected to the metaobject.

5.3 Metaobject Attachment

Attaching to objects: The MJVM uses an object store with object handles. A handle contains a pointer to the object's data and a pointer to the object's class.

When a metaobject is attached to an object, a shadow class is created and the class link of the object is redirected to this shadow class.

Attaching to references: When a metaobject is attached to a reference, the object handle is copied and the class pointer in the handle is changed to point to a shadow class. After copying the handle, it is no longer sufficient to compare handles when checking the identity of objects. Instead, the data pointers of the handles are compared.

6 Related Work

There have been described several reflective systems with related goals and properties as MetaJava. However, non of the systems provided the performance and flexibility, the MetaJava has achieved with the techniques described in this paper. A system with strong emphasis on performance is OpenC++ [2]. OpenC++ supports class based reflection. A method declaration can be annotated in the base-level class and invocations of this methods are later reified. More flexible tailoring mechanisms are provided by CodA [14] and AL-1/D [16]. CodA tries to identify the basic building blocks of an object-oriented run-time system. Because CodA is based on Smalltalk, it focuses on message exchanges and separates the subsequent actions in a message exchange from each other. Al-1/D separates different views of a base-level system. One view is concerned with the language semantics, another with resource management, and so on.

7 Project Status and Future Work

We are working on several other applications of the MetaJava architecture. We implemented metaobjects for remote method invocation, replication, and active objects. Further projects for meta objects include support for security policies [18], concurrency control [17], and distribution configuration.

The ultimate goal of our work is making reflection an integral part of the programming model and support composition of meta-level systems. One major issue is to develop concepts to make the attachment of the meta objects to software modules configurable and to keep such configuration statements out of the functional code of the application program. This can only be achieved by providing language support for specifying information about base-level objects.

To keep the security and robustness of the Java system, one must be able to specify the rights of metaobjects.

Currently the MetaJava system allows to modify the object model and implement extended object models. Further development will aim at providing the meta system access to core operating system facilities - for example, by extending the JavaOS operating system.

Information about the current project status is available at
http://www4.informatik.uni-erlangen.de/metajava/.

8 References

1. C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. Thesis. Stanford University. March 1992.
2. S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *ECOOP '93*, Kaiserslautern, Germany, LNCS 707, Springer-Verlag, pp. 482–501.
3. J. Ferber. Computational Reflection in class based Object-Oriented Languages. *OOPSLA '89*, New Orleans, La., Oct. 1989, pp. 317–326.
4. J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. *Proc. of the 25th IEEE Symp. on Fault Tolerant Computing Systems*, 1995.
5. Michael Golm, Jürgen Kleinöder. *Implementing Real-Time Actors with MetaJava*, Tech. Report TR-I4-97-09, Universität Erlangen-Nürnberg: IMMD IV, Apr. 1997
6. W. L. Hürsch, C. V. Lopes. *Separation of Concerns.* Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
7. G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, Vol. 13, No. 1, Jan. 1996, pp. 8-11.
8. G. Kiczales et al. *Aspect-Oriented Programming*. Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, June 14-15, 1996 (http://www.parc.xerox.com/spl/projects/aop/).
9. J. Kleinöder, M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. *IWOOOS '96*, Oct. 27-18, 1996, Seattle, Washington, IEEE, 1996.
10. J. Kleinöder, M. Golm. *Transparent and Adaptable Object Replication Using a Reflective Java*, Tech. Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996
11. M. Golm, J. Kleinöder. *MetaJava—Design and Implementation of a Platform for Adaptable Operating-System Mechanisms*, Tech. Report TR-I4-97-10, Universität Erlangen-Nürnberg: IMMD IV, Jun. 1997
12. T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.
13. P. Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.
14. J. McAffer. Meta-Level Architecture Support for Distributed Objects. *IWOOOS '95*, Lund, Sweden, IEEE, 1995, pp. 232–241.
15. P. Mulet, J. Malenfant, P. Cointe. Towards a Methodology for Explicit Composition of MetaObjects. *OOPSLA '95*. pp. 316-330
16. H. Okamura, M. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. *International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, LNCS 742, Springer-Verlag, Nov. 1993.
17. S. Reitzner. *Splitting Synchronization from Algorithmic Behavior*. Technical Report TR-I4-08-97, University of Erlangen-Nürnberg, IMMD IV, April 1997.
18. T. Riechmann. *Security in Large Distributed, Object-Oriented Systems.* Technical Report TR-I4-02-96, University of Erlangen-Nürnberg, IMMD IV, Mai 1996.
19. F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, Dec. 1990, pp. 299-319.
20. B. C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. Thesis, MIT LCS TR-272, Jan. 1982.
21. Sun Microsystems. *Java Core Reflection, API and Specification.* February 4, 1997.