# MetaJava - A Platform for Adaptable Operating-System Mechanisms

Michael Golm, Jürgen Kleinöder

April 1997 TR-I4-97-10

Technical Report

Computer Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Presented at the ECOOP 97 Workshop on Object-Orientation and Operating Systems, June 10, 1997, Jyväskylä, Finland

# MetaJava - A Platform for Adaptable Operating-System Mechanisms

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

## Abstract

*Fine-grained adaptable operating-system services can not be implemented with todays layered operating-system architectures. Structuring a system in base level and meta level opens the implementation of operating-system services. Operating-system services are implemented as meta objects and can be replaced by other meta objects if the application object 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 persistence and replication.
- The architecture must be general, that means several problems, such as persistence, distribution, replication, 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 systems. Section 5 shows the problems that had to be solved during 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 that support run-time mechanisms, such as multithreading (synchronization, dead lock detection, etc.), distribution, fault tolerance, mobile objects, extended transaction models, persistence, and so on. Many ad hoc extensions to languages and run-time systems have been implemented to support some of these mechanisms. But this solution does not meet the very diverse demands, different applications make on such mechanisms. Instead it is desirable to provide an application with means to control the implementation of its run-time mechanismans 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 [Smi82] was the first that considered reflection as essential part of the programming model. Maes [Mae87] 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 the execution of the functional code. To enable this supervision, some aspects of the base-level computation must be reified. *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 [KlG96].

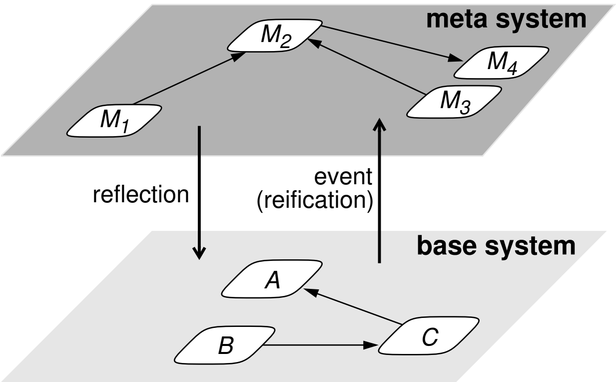
As pointed out in [Fer89] there are two types of reflection: structural and behavioral reflection (in [Fer89] termed computational reflection). Structural reflection reifies structural aspects of a program, such as inheritance and data types. The Java Reflection API [Sun97]is an example of structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. The main focus of MetaJava is to provide behavioral reflection capabilities. This is described in the next section.

## 3 Computational model of MetaJava

Traditional systems consist of an operating system and, on top of it, a program which exploits the OS services using an application programmer interface (API).

Our reflective approach is different. The system consists of the OS, the application program (the *base system*), and the *meta system*. The program may not be aware of the meta

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




The computations of objects A,B,C raise an event that transfers control to the meta level.

Figure 1: Computational model of behavioral reflection

system. The computation in the base system raises events (see Figure 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. 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 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.

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

Figure 2: Events generated by base computation

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 arbitrary 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 ref-

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.

Figure 3: Selected methods of the meta-level interface of the MetaJava virtual machine

erence, or when attaching a certain security policy to the reference [Rie96]. 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[2]. A list of the most important methods of the MLI is given in Figure 3.

## 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.

If several applications are exectuing within one Java machine, application-specific metaobjects lead to a hierarchy of metaobjects (Figure 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 [KlG96b], just-in-time compilation, active objects [GoK97], asyncronous 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

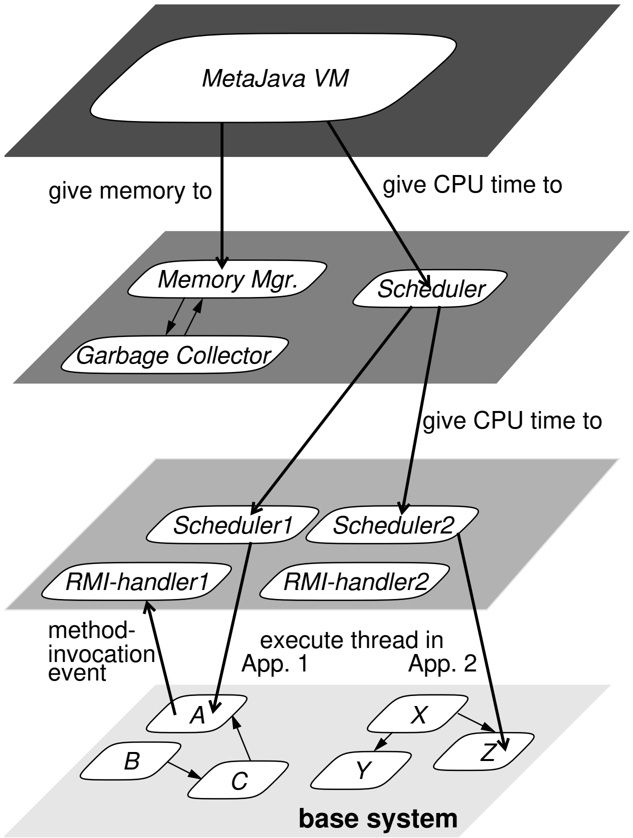



Figure 4: Hierarchy of meta levels

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 (Figure 3).

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

This section discusses the changes to the JVM that were necessary to enable our reflective model.

### 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. [Fer89] 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 [Fer89] 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

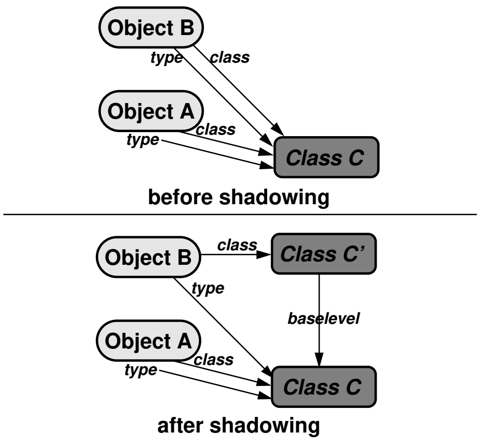
2. Architecture and terminology of the Java VM are described in detail in [LiY96]

classes. We introduce shadow classes to solve this problem which is inherent in class-based languages. A class-based language assumes that there usually exist many objects with the same type (class). Prototype based languages like Self [Cha92] or Moostrap support one-of-a-kind objects [Mul95]. It is easy to derive an object from another object and change the fields or methods of the new object without affecting the original[3]. MetaJava's shadow classes are an approximation of this behavior.

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. This makes the shadowing transparent to the base system. Several problems had to be tackled to ensure this transparency.




Objects A and B are instances of class C. A shadow class C' of C is created and installed as class of object B. Base-level applications can not differentiate between the type of A and B, because the type link points to the same class. However, A and B may behave differently.

Figure 5: The creation of a shadow class

**Class data consistency.** The consistency between C and C' must be maintained. All class-related non-constant data must be shared between C and C', because shadow classes are targeted at changing object-related, not class-related properties. The MJVM solves this problem by sharing class data (static fields and methods) between shadow classes and the original class.

**Class identity.** Whenever the MJVM compares classes, it uses the original class pointed to by the *type* link ( Figure 5). This class is used for all type checks, for example

- checking assignment compatibility
- checking if a reference can be cast to another class or interface (checkcast opcode)
- protection checks

**Class objects.** In Java classes are first class objects. Testing the class objects of C and C' for equality must yield true. In the MJVM every class structure contains a pointer to the Java object that represents this class. These objects are different for C and C', because the class objects must contain a link back to the class structure[4]. Thus, when comparing objects of type class, the MJVM really compares the type links.

Because classes are first class objects it is possible to use them as mutual exclusion lock. This happens when the monitorenter/monitorexit bytecodes are executed or a synchronized static method of the class is called. We stated above that shadowing must be transparent to the base level. Therefore the locks of C and C' must be identical. So the MJVM uses the class object of the class structure pointed to by the type link for locking.

**Garbage Collection.** Shadow classes must be garbage collected if they are no longer used, i.e. when the metaobject is detached.The garbage collector follows the baselevel link to mark classes in the tower of shadow classes (tower of metaobjects). Shadowed superclasses are marked as usual following the superclass link in the shadow class.

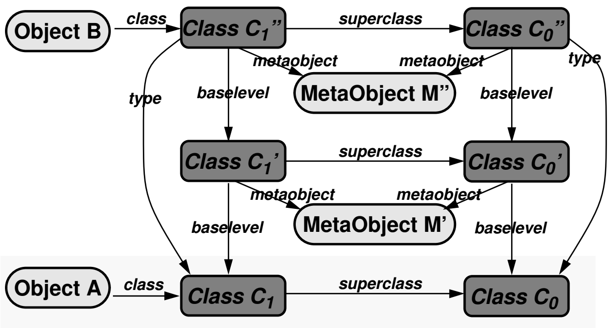
**Code consistency.** Some thread may execute in a base-level object of class C when shadowing and modification of the shadow takes place. The system then guarantees that the old code is kept and used as long as the method is executing. If the method returns and is called the next time, the new code is used. This guarantee can be given, because during execution of a method all necessary information, like the current constantpool or a pointer to the method structure, are kept on the Java stack as part of the execution environment. Experience will show if the code modification support, as currently provided by MetaJava, affects the robustness of the system.

**Memory consumption.** A shadow class C' is a shallow copy of C. Only method blocks are deep copied. A shallow class has a size of 80 bytes. A method has a size of 52 bytes. An entry in the methodtable is a pointer to a method and has a size of 4 bytes. Hence, the cost of shadowing is a memory consumption of about (80+number_of_methods*(52+4)) bytes per shadow class in MetaJava.

3. In fact, this is the basic mechanism of reuse in prototype-based languages.

4. In a different implementation the class object could also contain the complete class structure as object data.

**Inheritance.** During shadow class creation a shadow of the superclass is created recursively. When the object reverses to its superclass with a call to super the shadowed superclass is used. All shadowed classes in the inheritance path use the same metaobject.




The shaded area marks the original configuration. Object A and B are of class $C_1$ which is a subclass of $C_0$. When the metaobject M' is attached to object B, a shadow class $C_1$' of $C_1$ is created. Later, the metaobject M" is attached to B. This causes the creation of shadow class $C_1$".

Figure 6: Metaobjects and inheritance

**Original behavior.** Additional to the metaobject link a shadow class in MetaJava also needs a link baselevel to the original class. It is used when resorting to the original behavior of the class. In all non-shadow classes the baselevel link is null. The baselevel link realizes the *tower of metaobjects* (Figure 6). However, there is an important difference between the tower of metaobjects in MetaJava and the traditional notion of a tower of metaobjects. Traditionally, a tower of metaobjects is constructed so that each metaobject reifies structure and behavior of the metaobject one level below. In the MetaJava tower, a metaobject can continue (delegate) work to the base level if it has finished its own computations.

## 5.2 Implementation of the event mechanism

We are now able to transparently modify the class structure of individual objects. This section explains, how this mechanism can be used for a reification of object behavior.

**Reification of incoming messages (method invocations).** We talk about reification of incoming messages of object O (O is of class C), if the invocation of O.m() by object X is handled by the metaobject of O. To implement reification of method invocations we investigated the following alternatives:

(1) All interesting bytecodes (invokevirtual and invokespecial) are replaced by reifying pedants (e.g. invokevirtual_reflective). This is a clean way to create reflective objects but requires an extension of the bytecode set.

(2) The interpretation process of interesting bytecodes is extended.

(3) Method bytecodes of C.m() are replaced by a stub code which jumps to the meta space.

The third approach has the advantage of taking only local effects (unlike 1.) and avoiding a change of the interpreter fetch/decode loop (unlike 2.), thus possibly also working with Java processors. Therefore we decided to reify method invocations with stub generation at run time.

The metaobject that implements a specific method invocation needs of course access to the method arguments. An initial version of MetaJava used an object array to store method arguments. This was fast, but excluded primitive types as method arguments. In the current MJVM arguments are passed in a special argument container which is able to change the type of one slot dynamically.

**Reification of outgoing messages.** An outgoing messages is caused by the invokevirtual opcode[5]. The invokevirtual opcode has three arguments: a class name, a method name and a method signature. When registering for the event, the class name, method name, and signature can be specified. The code generator then looks for all invokevirtual opcodes that match, and creates the following code in place of the opcode:

(1) create event object with class name, method name, and signature

(2) create and initialize the argument object (TypeAdaptiveContainer)

(3) invoke the event dispatch function

While for reifying incoming messages the complete method at the target is replaced by a stub code, we now do code injection at the origin of the method call.

**Reification of instance variable accesses.** Reification of global accessible instance variables would either restrict the MJVM implementation to use an access function for every variable access or lead to modification of every piece of code that possibly accesses the variable. For this reason only accesses to variables with protection private protected can be reified in MetaJava. Variables with this protection can only be accessed in the declaring class or in its subclasses. But this also means that the variable can be accessed outside the current object in a different object of the same class, because protection in Java is not object based but class based. If this other object already uses a shadow class, chances a very bad to detect all putfield/getfield opcodes that are allowed to access this particular variable. To solve the problem we suggest a new object-based

5. The other opcodes that invoke methods are invokespecial to call constructors and invokestatic to call class methods. They are not interesting for our current considerations.

variable protection and two new opcodes putfield_this/ getfield_this. For now, we ignore the problem (i.e., we rely on programmer convention).

The stub generator for variable accesses of a variable with name N and type T in an object O of class C works as follows: Find all putfield and getfield opcodes that access a variable of the class C, name N, and type T. If the opcode is a putfield, remove the putfield and insert a stub code with the following functionality:

(1) Create and initialize an EventDescFieldAccess object. The object contains information if the access was reading or writing. If it was writing, the event-description object also contains the intended new value of the variable.

(2) Invoke the static eventDispatchVoid function of the class MetaObject with the event object as parameter.

Relevant getfield opcodes are replaced by a similar stub.

**Reification of object locking.** For some base-level mechanisms it is either not possible or prohibitive expensive to reify them with code injection. Object locking is one such mechanism. The stub generator would have to generate stubs for all monitorenter and monitorexit opcodes. Furthermore, all executions of synchronized methods of this object would have to be reified. But this does not guarantee that all accesses to the lock are reified at the meta level, because the JVM or native code libraries can directly access the lock.

Every JVM needs a function which maps the Java object lock to a system dependent lock. The MJVM uses a function pointer in the class of the object to acquire or release the lock. This function pointer normally refers to a function which implements locking and unlocking. When registering for the acquire-object-lock or release-object-lock events, the MJVM modifies this pointer to point to the metaobject's event handler. The responsible metaobject can now use arbitrary complex algorithms to manage the lock. If the metaobject decides to continue with the standard locking protocol it calls the lock function at the class one level below (following the baselevel link).

When handling object locking we must be careful not to make the pitfall of reflective overlap. Reflective overlap [Mae87] occurs, if the reflective compution influences the base-level explicitely and implicitely. To not implicitely influence the base-level computation, the meta-level event handler for lock L should not use any method that acquires the lock L.

**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 is interested in the mechanism it is delegated to this metaobject.

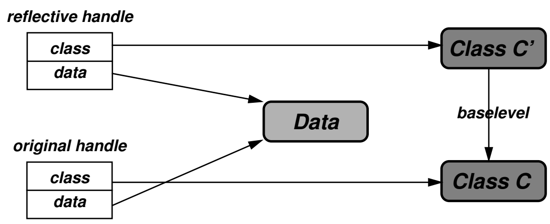
### 5.3 Metaobject attachment

The previous section explained, how it is possible to change the behavior of individual objects without affecting other objects of the same class. This section explains, how metaobjects can be attached to objects and references.

**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.

A metaobject can be used to control a number of base objects of the same class in a similar manner. It is not necessary to create a shadow class for every object. Shadow classes are thus installed in two steps. In the first step, a shadow class is created. In the second step, the shadow class is installed at the object. Once a shadow class is created, it can be installed at multiple objects.

**Attaching to references.** The current implementation of MetaJava uses an indirect object store. If a metaobject is attached to a reference, the semantics, i.e. class, of operations involving this reference must change.Therefor the handle is copied and the class pointer in the handle 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. This requires that data pointers are unique, i.e. that every object has at least a size of one byte. The MetaJava object store guarantees this.



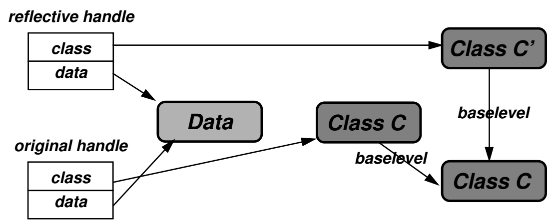

This figure shows how a shadow class is connected to a reference. A reference is represented by a handle, which contains a link to the object's data and a link to the object's type (class or methodtable). If a shadow class must be attached to the handle (original handle), the handle is cloned (reflective handle) and the class link is set to the shadow class/

Figure 7: Attaching to a reference

**Multiple Binding.** We mentioned that it is possible to attach more than one metaobject to a base-level object. The most recently bound metaobject is activated first. If this metaobject decides to continue with the default base-level action, the next lower-level metaobject in the tower of metaobjects is activated. If one metaobject does not continue with the default action, lower metaobjects take no effect.

It is obvious from this description, that the order of metaobjects in the tower is important. An example may illustrate this. First the user decides to trace method invocations and attaches a tracing metaobject T to the base object B. At a later point in execution it becomes necessary to replicate the base object and thus an actively replicating metaobject R is attached to B. If now a method of B is called, control is first transferred to R, which replicates the method call and initiates control transfer to the base level at every replica. Now the event handler of metaobject T is invoked and outputs a trace message at every node. If the metaobjects are attached in the opposite order, the trace is performed before the method call is replicated.

The situation is a bit more involved if the metaobjects are attached to an object and a reference to the object. The MJVM allows to attach a metaobject first to a reference and then to the object (using this reference), but the semantics of object attachment would be different from the behavior a programmer might expect.




This figure shows the situation after a metaobject is attached to a reference and then to the object this reference refers to.

Figure 8: Attaching to a reference

The metaobject attached to the object receives only those events, that are related to the reference.

## 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++ [ChM93]. 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 customization mechanisms are provided by CodA [McA95] and AL-1/D [OIT93]. 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 [Rie96], concurrency control [Rei97], distribution configuration[Bec97].

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


Bec97 U. Becker. Beschreibung von Verteilung im objektorientierten Design einer Anwendung. Diplomarbeit Informatik. Universität Erlangen, 1997.

Cha92 C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD. Thesis. Stanford University. March 1992.

ChM93 S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceedings of ECOOP '93, the 7th European Conference on Object-Oriented Programming,* Kaiserslautern, Germany, LNCS 707, Springer-Verlag, pp. 482–501.

Fer89 J. Ferber. Computational Reflection in class based Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '89,* New Orleans, La., Oct. 1989, pp. 317–326.

FNP+95 J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. *Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems,* 1995.

GoK97 Michael Golm, Jürgen Kleinöder. *Implementing Real-Time Actors with MetaJava,* Tech. Report TR-I4-97-09, Universität Erlangen-Nürnberg: IMMD IV, Apr. 1997

HüL95 W. L. Hürsch, C. V. Lopes. *Separation of Concerns.* Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

Kic96a G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software,* Vol. 13, No. 1, Jan. 1996, pp. 8-11.

Kic96b 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/).

KlG96 J. Kleinöder, M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. *Proc. of the International Workshop on Object Orientation in Operating Systems - IWOOOS '96*, October 27-18, 1996, Seattle, Washington, IEEE, 1996.

KlG96b Jürgen Kleinöder, Michael Golm. *Transparent and Adaptable Object Replication Using a Reflective Java*, Tech. Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996

LiY96 T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.

Mae87 P. Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.

McA95 J. McAffer. Meta-Level Architecture Support for Distributed Objects. *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems,* Lund, Sweden, IEEE, 1995, pp. 232–241.

Mul95 P. Mulet, J. Malenfant, P. Cointe. Towards a Methodology for Explicit Composition of MetaObjects.Proc. of OOPSLA '95. pp. 316-330

OIT93 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.

Rei97 S. Reitzner. *Splitting Synchronization from Algorithmic Behaviour.* Technical Report TR-I4-08-97, University of Erlangen-Nürnberg, IMMD IV, April 1997.

Rie96 T. Riechmann. *Security in Large Distributed, Object-Oriented Systems.* Technical Report TR-I4-02-96, University of Erlangen-Nürnberg, IMMD IV, Mai 1996.

Sch90 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.

Smi82 B. C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. Thesis, MIT LCS TR-272, Jan. 1982.

Sun97 Sun Microsystems. *Java Core Reflection, API and Specification.* February 4, 1997.