Design and Implementation of a Meta Architecture for Java

Michael Golm

Jan 1997 DA-I4-02-97

Diplomarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)

Design and Implementation of a Meta Architecture for Java (Entwurf und Implementierung einer Meta-Architektur für Java)

Diplomarbeit im Fach Informatik

vorgelegt von

Michael Golm

geb. am 05.10.1970 in Leipzig

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. F. Hofmann*** ***Dr. Jürgen Kleinöder***
Beginn der Arbeit:	01.08.1996
Abgabe der Arbeit:	31.01.1997

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 31.01.1997______________________________

Author's Declaration

Some of the material presented in this thesis has been previously published in the following documents:

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

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

Abstract

Adaptability to special requirements of applications is a crucial concern of modern operating and run-time system architectures. Reflection and meta architectures are means to achieve this adaptability.

This thesis describes an extension of the Java virtual machine that allows flexible and selective attachment of reflective properties to objects.

The extended machine - MetaJava - supports structural and behavioral reflection. Its structural reflection capabilities allow the meta-level system to gather information about the base-level program structure. Behavioral reflection transforms behavior into events. Relevant base-level behavior, such as method invocations, instance variable accesses, and object locking, is reified by the delivery of events to a metaobject.

This thesis explains all changes necessary to the Java virtual machine to support run-time reflection with a minimal penalty in run-time performance. Run-time code generation is used to avoid a performance degradation of non-reflective parts of the system.

The MetaJava architecture does not limit meta-level applications to special domains, such as distributed or concurrent computing, but can also be used to implement different security policies, just–in-time compilation, or location control of mobile objects. A remote method invocation mechanism, an object replication system and a just-in-time compiler are described to demonstrate how the Java environment can be enhanced using our meta architecture.

Zusammenfassung

Adaptierbarkeit an spezielle Anforderungen von Anwendungen ist ein wichtiges Anliegen moderner Betriebssystemarchitekturen und Laufzeitumgebungen. Reflexion und Metaarchitekturen sind Mittel, um diese Adaptierbarkeit zu erreichen.

Diese Diplomarbeit beschreibt eine Erweiterung der Virtuellen Java Maschine, die flexibles und selektives Binden reflexiver Eigenschaften an Objekte ermöglicht.

Die erweiterte Maschine - MetaJava - unterstützt strukturelle Reflexion und Reflexion über das Verhalten von Berechnungen. Strukturelle Reflexion erlaubt dem System auf Metaebene, Informationen über die Struktur des Programms auf der Basisebene zu erlangen. Verhaltensreflexion geschieht durch die Umwandlung des Programmverhaltens in Ereignisse. Relevantes Basisebenenverhalten, wie zum Beispiel Methodenaufrufe, Zugriff auf Instanzvariablen und das Sperren und Freigeben von Objekten, wird durch das Ausliefern von Ereignissen an ein Metaobjekt reifiziert.

Die Diplomarbeit beschreibt alle Änderungen an der JVM, die nötig sind, um Reflexion zur Laufzeit mit einer minimalen Beeinträchtigung der Systemleistung zu erreichen. Kodegenerierung zur Laufzeit wird benutzt, um einen Leistungsabfall von nicht-reflexiven Teilen der Maschine zu vermeiden.

Die MetaJava Architektur schränkt Anwendungen der Metaebene nicht auf spezielle Gebiete ein, wie verteiltes oder nebenläufiges Rechnen, sondern kann auch benutzt werden, um verschiedene Sicherheitsstrategien, Just-in-time Compiler oder ein System zur Steuerung des Ortes mobiler Objekte zu implementieren. Um zu demonstrieren, welche Verbesserungen MetaJava gegenüber Java bietet, werden ein Mechanismus zum Aufrufen entfernter Methoden, ein Objekt-Replikationssystem und ein Just-in-time Compiler beschrieben.

Introduction

1. Problem statement

Today's operating systems and software engineering practice are insufficient to cope with the demands of modern applications in adequate ways. Modern software systems should be adaptable to changing run-time environments and to changing application requirements at minimal cost.

Adaption of applications to changing run-time environments

Current software systems act in an environment which is constantly changing. This changes could simply be caused by the introduction of new hardware or the use of the system in an environment that was not anticipated at development time. Most applications are not prepared to take advantage of the new features of the hardware or the new environment.

Adaption of programs to changing application requirements

Modern software systems are faced with a variety of new requirements. Consequently they demand support from the operating and run-time system for a variety of services. Operating systems tried to tackle these demands with the inclusion of new services, like persistence (Grasshopper [Dearle et al. 94]) or fault-tolerance (Delta-4 [Powell 91]).

Operating systems are traditionally structured in a kernel space and a user space. The kernel controls the hardware, maintains resources, and offers services through a system call interface. The kernel can provide some services transparently to the application program, for example virtual memory support. This transparency is the advantage of a kernel based approach to new services. However, the inclusion of new services into the operating system has some drawbacks. Integrating a new service in an operating system or changing an existing service is an expensive and time consuming task, which usually can be done only by the operating system vendor, not by application developers. Microkernel operating systems promised to ease the task. But it still requires an experienced systems programmer to integrate a new service into an operating system environment. Furthermore it is not possible or not easy to adapt the operating system support on a per application basis.

Therefore most applications rely on libraries to provide commonly used services and programmers develop their applications explicitly using those libraries. Libraries can be used and adapted more flexible than operating system services. However, libraries can not provide services transparently.

Reflection as a new structuring technique

Splitting large and complex systems into smaller units is common engineering practice, not only in software engineering. In software engineering there is a trend to more and more sophisticated separation mechanisms. At the beginning, code was collected into functions, then functions were grouped into modules, then into objects and classes. This thesis explores the next step in this sequence - the separation of objects according to their self referential or reflective behavior. Some of the previously mentioned new application requirements can only be satisfied if the system observes and modifies its own behavior, i.e. reflects about its own computation. When using separation in base level and meta level, the reflective and non-reflective parts of a program can be developed independently. Like object-oriented programming, reflection and metaprogramming are new paradigms that lead to more structured and more easily maintainable programs. This leads to an increased programmer productivity.

2. Thesis goals

Java is a safe, object-oriented programming language. The goal of this thesis is to build a reflective architecture for the Java environment. The reflective architecture should exhibit the following properties:

- It must be possible to separate functional and non-functional code. In conventional programming the application program is mixed with and complicated by policy algorithms. This makes it difficult to understand, maintain, debug, and validate the program. Separation of the reflective from the base algorithm makes reusability of policies feasible [Hürsch & Lopez 95]. The use of a separate meta space allows the application programmer to focus on the application domain. The additional functionality is supplied as a library of meta components or is developed parallel to the application program.
- Separating functional and non-functional code makes both parts of the system reusable. Furthermore, it must be possible to replace non-functional, if the system must be adapted to a new run-time environment. A change of the meta-base relation should not impose changes of base-level class code. This implies that the connection between base-level objects and meta-level objects must be specified outside the base-level class definition.

3. Historical comments

about 1945

The computer architecture invented by John von Neumann stores code and data in the same memory. This makes it possible to treat code as data. First use of this architectural property was dynamic linking, adjusting jumps when moving code. Modifications of programs by themselves (“self modifying code”) was soon considered bad programming practice, partly because no formal model was found as a basis for such code modifications.

1972..1980

Smalltalk [Goldberg & Robson 89] has been developed. Smalltalk uses meta classes to store information about classes and treats classes as first-class objects that are instances of a meta class.

1981

[Foderaro & Fateman 81] report performance increases when they give the virtual memory subsystem hints about the planned memory usage in a Smalltalk mark-sweep garbage collector.

1982

Brian C. Smith [Smith 82] develops 3-Lisp, a reflective dialect of Lisp. 3-Lisp is the first language that considers reflection as an important language concept.

1984

[Friedman & Wand 84] separate the concept of reification from the concept of the tower of interpreters.

1984

The production rule system SOAR is developed. It is able to inspect and modify the inference engines internal data. Production rule systems like SOAR or OPS5 apply rules to data to modify this data. Rules are activated (fired), if their condition on the left-hand-side is fulfilled. These systems must implement a conflict resolution strategy, which is used, when more than one left-hand-side is fulfilled. SOAR can modify this conflict resolution strategy.

1987

Pattie Maes develops 3-KRS to demonstrate that it is feasible to construct a reflective object-oriented language.

1991

Kiczales, Rivieres and Bobrow develop the CLOS Metaobject protocol.

4. Outline

This thesis consists of four chapters. The first chapter serves as an introduction to the concepts of reflection and meta architectures. We define basic terms and describe prominent reflective architectures and the concepts related with them. The second chapter presents the design and implementation of MetaJava. The structure of a Java virtual machine is explained and the changes that are necessary to enable reflective computation. The third chapter illustrates the benefits of the meta-level extension. We explain how to develop a meta-level application within the MetaJava environment. Finally, Chapter 4 contains a conclusion and suggests future work.

Chapter I
Meta-level Architectures and Reflection

1. Object orientation

Object-oriented programming and object-oriented concepts have faced massive scepticism some years ago but are now well established in main-stream software development.

There seem to be several reasons for this success. Object-oriented analysis and object-oriented design are well suited to manage complexity in large systems by splitting the system in smaller units and providing powerful mechanisms to combine them. The use of polymorphism and abstract data types allows the designer to model a problem using clear abstractions instead of exact but complicated views of reality. However, it became obvious that object-oriented techniques have difficulties with separating functional and non-functional aspects of an algorithm. Reflection and meta programming are means to achieve this separation.

2. Reflection and meta programming

In the past programs had to fulfil a task in a limited computational domain. Demands of today's applications are becoming more complex: multithreading and problems of synchronization and deadlock detection, distribution, fault tolerance, mobility, extended transaction models, persistence, and so on. These demands make it necessary for an application program to observe and adjust its own behavior. Many ad hoc extensions to languages and run-time systems have been implemented to support the above mentioned features. Reflection is a fundamental concept for a uniform implementation of all these different demands.

We will now introduce some terms to enhance the lucidity of the following discussion.

Def. I.1 **Computational System, Computational Domain**

A computational system is a system that uses a specific algorithm to change its computational domain according to a certain specification.

Def. I.2 **Reflection**

Reflection is the capability of a computational system to “reason about and act upon itself” [Maes 87a] and adjust itself to changing conditions. The computational domain of a reflective system is the structure and the computations of the system itself.

Def. I.3 **Reification**

Reification is the act of making something accessible which normally is not available in the programming model or hidden from the programmer.

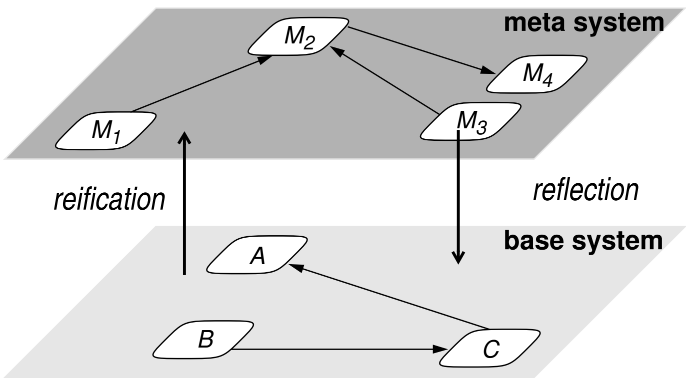
Def. I.4 **Base System**

The base system is a computational system which is the domain of a different computational system, the meta system.

Def. I.5 **Meta System**

The meta system is a computational system whose domain is another computational system, the base system. A Meta-System is located behind the base system and is normally invisible to the same.

The relation between reflection, reification, base system, and meta system is illustrated in Figure I.6.



Objects A, B, and C solve a specific application problem. Meta-level objects $M_{1..4}$ control the execution of objects A,B,C and implement parts of their behavior.

Figure I.6 Reflection and reification in a computational system

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.

Def. I.7 **Meta-level object**

A meta-level object is an object that is located at the meta level and implements meta-level functionality.

Def. I.8 **Metaobject**

A metaobject is an object that contains information about another object (the base-level object) and/or controls the execution of this object.

Metaobjects are meta-level objects modeling one base-level object. Note that not all meta-level objects are metaobjects, but all metaobjects are meta-level objects.

Transparency and orthogonality of base system and meta system are desirable features but cannot always be guaranteed in real applications. Consider a metaobject that implements a bounded buffer synchronization scheme. It does not make sense to use it with a base object that does not have a bounded buffer semantics (i.e., two operations *put* and *get* to access the buffer). Therefore the MetaJava system enables the developer who combines base and meta level to build applications, to pass semantic information about the base object to the metaobject.

3. Meta-circular interpreters

The concept of reflection has its offspring in artificial intelligence research and language theoretic research. Most of these languages were derivatives of the functional programming languages Lisp and Scheme, for example 3-Lisp [Smith 82], Brown [Friedman & Wand 84], Blond [Danny & Malmkjær 88], Refci [Simmons et al. 92]. The intention of most functional reflective languages is the introduction of a general mechanism to extend the language and eliminate special forms. Separation of concerns and good performance were only of minor interest. The prevailing technique of implementing reflection in functional languages is the meta-circular interpreter.

Def. I.9 **Meta-circular interpreter**

An interpreter that is written in the language it interprets (host language), using facilities of the host language.

When some levels of these interpreters are written, they form a reflective tower of interpreters.

The central theme in the work on reflection in Lisp and Scheme based languages was the reification of environments and continuations. Functional programs are using an *environment* when evaluating a function. The environment is a mapping from names to values and contains the current bindings of variables. The environment changes when a function executes such *special forms* as `setq`. Special forms are functions provided by the interpreter. In conventional Lisp systems they can not be replaced by user defined functions, partly because the user has no means to manipulate the environment directly. Reflective functions provide the user with such means, thus eliminating the need for a lot of special forms and making it possible to construct minimal and extensible Lisp systems.

Using the approach of the meta-circular interpreter for MetaJava would mean writing a Java interpreter in Java (J-JVM) and running all programs with this interpreter. The interpreter itself would be executed by an interpreter written in C (C-JVM). The self representation would then be available as Java objects, the causal connection would automatically be guaranteed, and the system would be fully reflective (i.e., any aspect of the system could be modified). Every time a reflective computation is started, interpretation steps up to the executing interpreter. If it steps up to the level of the C-JVM, a new J-JVM is created from the self representation of the C-JVM. This is the conventional way of building a tower of meta-circular interpreters. However, we believe that it is not necessary to have a complete self representation available. Chapter II.3.1 will list the parts of the interpreter that are included in MetaJava's self representation. This limitation avoids the problems of a meta-circular interpreter. First there is a performance problem. Every meta-circular interpreter would run some orders of magnitude slower than the interpreter one level above. Furthermore, the self representation of the system would probably depend on implementation details of the J-JVM.

4. Categorizing reflection

We now try to identify criteria for a categorization of reflective systems.

4.1 Compile-time vs. load-time vs. run-time reflection

As with most other activity the *whence* is also an important criteria in meta interaction. The life of a program can be separated in compile time, link time, load time, and run time [1], depending on the time when the binding between base and meta system is established. We only consider compile time, load-time, and run time as possible alternatives for meta-level interactions. As usually there is a trade-off between flexibility and efficiency. While at an early time efficiency is high and flexibility is low, the contrary applies to reflection at late times.

1. And probably a lot of other times depending on the definition of a programs life cycle.

4.1.1 Compile-time reflection

While compiling a base-level program the compilation of certain language expressions is controlled by a meta-level program with a meta-level program modifying the compilation process.

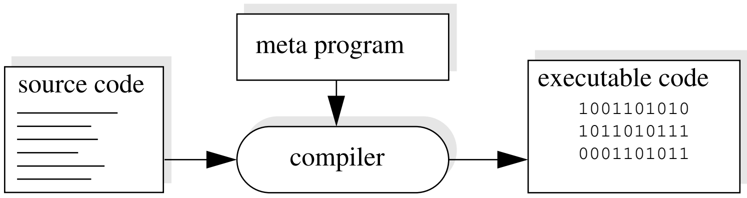


Figure I.10 Compile-time meta architecture

An advantage of compile-time reflection is the run-time efficiency. Overhead occurs only at compile time. As mentioned above, performance is traded for flexibility.

Compile time meta-level architectures change the way a program is translated from source format to binary format. With compile-time reflection it is possible to modify the syntax as well as the semantics of a programming language. Compile-time reflection usually operate in terms of syntactical entities like classes and types, but can not handle run-time entities, like objects, very well.

The first most unflexible kind of compile-time reflective architectures are annotation based systems, where the set of available annotations can not be extended and the semantic of the annotations is fixed. There are two very common mechanisms of these annotation systems, C pragmas and High Performance Fortran (HPF) pragmas. In HPF the distribution of matrixes over processors can be specified with pragmas.

A more flexible way is, to allow the programmer to define own annotations. Two examples are C macros and Anibus [Rodriguez 91]. A C macro can be considered a programmed annotation. The macro itself is kind of a simple metaprogram, which computes over its arguments and generates C code. Anibus [Rodriguez 91] is an extension of MarkedScheme. It allows to define new marks which are small metaprograms.

The best flexibility at compile-time provide systems with a complete meta architecture. An example is OpenC++ (cf. Section 6.1.3). The compilation can be customized completely because the parse tree is available to the meta-level program.

4.1.2 Load-time reflection

Load-time reflection is done when loading and linking an application before executing it.

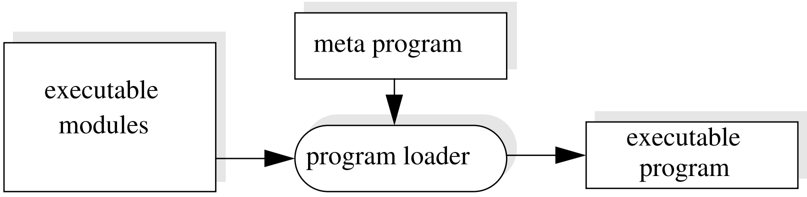


Figure I.11 Load-time meta architecture

The OMOS [Orr 95] system is an example for a load-time reflective architecture. The main focus of OMOS is in the configuration of programs by module composition. The system uses a special language to describe the composition of modules. The OMOS system can be seen as a very powerful dynamic linker. When using load-time reflection overhead exists only at application start-up time. The disadvantage is the lacking adaptability to run-time behavior [2].

4.1.3 Run-time reflection

As the name indicates, a run-time meta architecture primarily operates at run time.

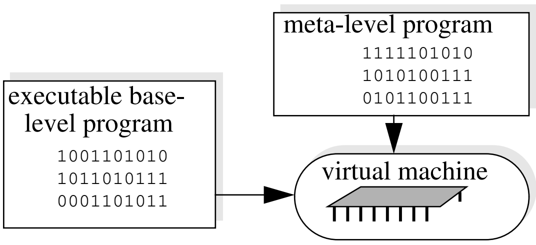


Figure I.12 Run-time meta architecture

Figure I.12 sketches the operation of a run-time meta architecture. The execution of a base-level program is modified according to the specification of the meta-level program. Advantages of run-time meta architectures are high flexibility and the possibility to adapt the run-time behavior of the base-level program. A disadvantage is the overhead incurred by reflective computation at run-time. Examples for run-time meta architectures are MetaJava and GARF (cf. Section 6.1.7).

4.1.4 Why compile-time reflection is not enough?

MetaJava uses run-time reflection, because we regard it necessary to change object behavior at run time and to attach metaobjects to arbitrary objects at run time. The identity of those objects is not known at compile time.

2. This does not apply to OMOS, because OMOS also employs run-time reflection.

There are several statements in the literature concerning the time of reflection. [Fabre et al. 95] complain about the limitations of the compile-time reflective language OpenC++. They miss a facility to attach metaobjects to base objects at run time. [Garbinato et al. 95] state that a static system, i.e. one that operates at compile time, would be enough. However, they use a dynamic system, which allows run-time reflection.

4.2 Structural vs. behavioral reflection

As pointed out in [Ferber 89] there are two types of reflection: structural and behavioral reflection (in [Ferber 89] termed computational reflection). Structural reflection reifies structural aspects of a program, such as inheritance, class structures, and data types. Run-Time Type Identification (RTTI) of C++ [Stroustrup 92] is an example of structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. A method invocation [3] is the most typical behavioral aspect of an object-oriented program.

4.3 Explicit vs. implicit reflection

A system uses explicit reflection if the transition from the base level to the meta level is visible at the base level. If the transition is not visible at the base level, reflection takes place implicitly.

5. Design issues in a reflective architecture

This section discusses some issues when designing an architecture that supports reflection.

5.1 Causal connection

The changes performed by the meta system at the self representation of the system must take effect to the base system. A change in the base system must induce a change of the base systems representation at the meta level. This connection between the base system and the meta system is called the *causal connection* [Smith 82]. A system which uses the self representation to actually implement the base system, as in meta-circular interpreters, has automatically a consistent self representation and fulfills the causal connection requirement.

5.2 Separation of concerns

Separation of concerns [Hürsch & Lopez 95] is the motivating force behind work on meta-level architectures. [Hürsch & Lopez 95] identify the concerns of class organiza-

3. Sometimes also called message send [Goldberg & Robson 89].

tion [4], process synchronization, location control, real-time constraints, failure recovery, and persistence. They also state their believe that the list of concerns is open-ended. [Hürsch & Lopez 95] separate between the basic concern and special purpose concerns, we called this functional and non-functional requirements.

5.3 Composition of concerns

A problem with meta-level architectures is the composition of concerns. If a system has only one meta-level, the concerns synchronization and replication, for example, are interwind at this level. We have for now no methodology how to compose different concerns. It seems trivial, if the concerns are completely orthogonal. But in practice, orthogonality is only an exception. More practical experience with meta-level programming in MetaJava will probably enlighten the problems related to metaobject composition.

5.3.1 Binding base and meta level

Base level and meta level must be connected in some way. Some architectures use one metaobject per base-level object and create the metaobjects on demand. Other systems provide a mechanism to dynamically bind a metaobject to a base-level object.

5.3.2 The problem of metaobject composition

To date meta-object composition seems to be an unsolved problem. The aim is to develop different meta-level functionalities, i.e. concerns, in different metaobjects. The composition of those metaobjects should yield a system which contains the combined functionalities. The main problem of metaobject composition is the semantic overlap of meta-level components. The non-orthogonality of metaobjects restricts composability.

Mulet et al. [Mulet et al. 95] discuss those problems and tries to provide a mechanism for metaobject composition. In their architecture metaobjects are divided into left-composed and right composed metaobjects. A left composed metaobject contains a link to another metaobject (aggregation), possibly the basic metaobject that does not change object semantics. The right-composed metaobject is used by another metaobject. Left-composed metaobjects must be "cooperative", i.e. they must let the right-composed participate in the meta-level computation with sending messages to it.

5.4 Reflective overlap

Relective overlap [Maes 87a] appears, if the meta level modifies a part of the system explicitly using reification and reflection, but also implicitly by its own computation. Reflective overlap can cause unpredictable results and thus must be avoided. The reifica-

4. To mention "class organization" as a special concern seems unusual at first glance. The separation of class structure from algorithms was suggested by adaptive programming (AP) research (cf. Section 7.3). The authors of [Hürsch & Lopez 95] belong to the AP community.

tion of the run-time stack can cause reflective overlap, because the meta system uses the reified stack for its own computation and thus modifies the stack implicitly and explicitly.

5.5 Recursive reflection

Recursive reflection occurs, if a metaobject is used as its own metaobject. [Kleinöder 92] describes the following example of recursive reflection. A metaobject implements remote method invocation. To resolve object names, it must communicate with a nameserver object. The nameserver is a remote object, and thus a remote method invocation must be used. To terminate the obvious recursion, the metaobject that implements the invocation of methods of the remote nameserver must be simpler than the general remote method invocation metaobject. The simple metaobject could be configured to use a specific host for method invocations.

6. Reflective systems

We now investigate how the concepts of reflection and metaprogramming are used in programming languages and operating systems. Then we describe some concepts that are related to reflection.

6.1 Programming languages

6.1.1 3-Lisp

3-Lisp [Smith 82] is an attempt to reconstruct the language Lisp within itself. 3-Lisp was the first language that incorporated reflection as an important programming paradigm. It exclusively uses explicit reflection. There exist two different kind of functions, non-reflective and reflective functions. If a reflective function is called, a level shift takes place and a complete self representation is available to the reflective function. The self representation consists of the environment and a continuation.

3-Lisp was followed by other systems based on Lisp or Scheme [Friedman & Wand 84] which introduced minor modifications but were only able to reify variable bindings.

As Smith states, his work is only concerned with reflection in programming languages, not in general computational systems (cf. [Smith 82] p. 451). Therefore he has no reason to think about the separation of concerns, because he has only one concern - the representation of a programming language. Separation of concerns is a central aspect in MetaJava.

6.1.2 3-KRS

3-KRS [Maes 87a] introduces reflection in prototype-based object-oriented systems with the concept of a metaobject. In 3-KRS every object has a metaobject. Normally this is the default metaobject, which implements the default semantics.

The base-level object has a *meta* link to its metaobject, the metaobject has a *referent* link to its base-level object. A metaobject in 3-KRS contains all information that is available about its base object. This information includes

- bookkeeping data (who created the object, when, etc.)
- how the object inherits information
- how it is printed
- how a new instance is created

There are some "triggers" that initiate a shift from the base level to the meta level:

- The base-level object mentions its metaobject.
- The metaobject overrides the default behavior of the base-level object.

The causal connection requirement is solved with the use of a meta-circular interpreter. A metaobject in 3-KRS is used for structural and behavioral reflection, i.e. it contains the structure of an object and implements it.

A object declaration in 3-KRS explicitly mentions the responsible metaobject. This makes reuse of metaobjects possible, but it seems difficult to reuse base-level objects with different metaobjects.

6.1.3 OpenC++

Building a meta-level architecture for C++ is not an easy task. Traditional C++ compilers limit the information that is available (i. e. inspectable and modifiable) at run time. Ideas to save information at compile time for use at run time are described in [Johnson & Palaniappan 93].

OpenC++ Version 2 [Chiba 96] is a compile-time metaobject protocol for C++. The OpenC++ compiler translates an OpenC++ program into a C++ program. The translation process can be customized by a meta-level program. OpenC++ is a very good tool for performing optimizations that a standard optimizer cannot carry out because an application-specific meta-level program uses information about program semantics that is usually not available to an optimizer. Due to its compile-time architecture, C++ has some limitations. It is only possible to create reflective classes but not reflective objects or references. Meta classes cannot be attached to base classes dynamically at run time. To reify method calls the compilation of the calling code is modified. Thus it is not possible to use reflective classes from precompiled code.

Due to its compile-time architecture OpenC++ only allows metaobject attachment to classes.

6.1.4 CodA

CodA [McAffer 95] uses a fine-grained decomposition of object behavior into fundamental components of base-level objects. These components are

- Send: Handles outgoing messages.
- Accept: Communicates with Send to accept or reject messages.
- Queue: Queues messages accepted by Accept but not yet processed by Receive.
- Receive: The Receive component is asked by to base object to receive the next message.
- Protocol: The Protocol component is asked by the base object to map a message to a method.
- Execution: The execution component is asked by the base object to execute a certain method.
- State: Knows about the state of the object.

CodA was designed to be used in a distributed environment and this is visible in the decomposition. For example receive, queue, and accept are mainly used to implement different message-queueing policies. CodA is a promising approach to a structured composition of meta-level components. CodA is an approach to “engineer the metalevel”[McAffer 96]. Different components of an object-oriented computational system are identified. The metalevel engineer composes these components to create new object behaviors. CodA is a meta architecture for Smalltalk, although the author claims that the principles are language independent.

The dominating aspect of computations in Smalltalk is the exchange of messages between objects. Consequently CodA deals exclusively with the reification and reflection of method invocations.

6.1.5 ABCL/R2

ABCL/R2 [Masuhara et al. 92] introduces the notion of a *hybrid group reflective architecture*. Metaobjects control object groups and the management of resources shared by the group members. The aim of ABCL/R2 is to model resource management at the meta level. ABCL/R2 uses two kinds of reflective towers: an individual tower like in 3-KRS and a group tower. The link between base and meta level is hardwired into the object definition.

Metaobjects are explicitly used in the script of the base-level object. There is a 1-to-1 relationship between objects and metaobjects (i.e., each object has one metaobject and each metaobject has one object). Metaobjects are created on demand, when they are explicitly accessed. The reflect statement is used to start a reflective computation.

6.1.6 AL-1/D

AL-1/D [Okamura et al. 92], [Okamura et al. 93] is a Smalltalk-based language for distributed computing. Interpreters on different hosts communicate with TCP/IP. AL-1/D partitions the meta level into different views of the base system to facilitate programming in a distributed environment. The views are possibly overlapping. A view is represented by a model, therefore the concept is called *Multi-Model Reflection Framework*. All models run concurrently. Sequential and concurrent consistency are addressed in [Okamura et al. 93]. The behavior of an object is represented by state transition diagrams with concurrent transitions. In AL-1/D the causal connection requirement is fulfilled with special retrieve and unretrieve bytecodes, that are inserted respectively at the begin and end of a method of a meta-level object.

The following models are available:

- Operation: The operation model is by far the biggest model of AL-1/D. It provides access to language semantics.
- Resource: The resource model provides access to the operating system (context,segment,scheduler,garbage collector).
- Statistics
- Distributed Environment
- Migration
- System

The partitioning contains a lot of predefined semantics. In MetaJava, active objects and message queues, for example, do not need to be part of the language's programming model, because they can be implemented at the meta level.

The performance overhead of the AL-1/D meta level is about 20 [Okamura et al. 93], which is comparable with our results (cf. Chapter II.5).

6.1.7 GARF

GARF [Garbinato et al. 95] is a reflective system for distributed computing. Smalltalk is used as base language. The authors describe two classes of metaobjects - which they call behavioral objects - encapsulators and mailers. Encapsulators reify method calls. Mailers are used to transport messages between encapsulators. GARF allows the installation of metaobjects at run time, which is similar to our approach. GARF was used to realizes fault-tolerant Smalltalk machines with an active replication system.

6.1.8 Moostrap

Moostrap [Mulet & Cointe 93] is a prototype-based language, influenced by concepts of the Self language. An object consists of data slots and function slots. A method invocation is split into two phases: the *lookup* of the slot associated with the method name and an *application* of this slot. Lookup and apply can be customized independently. With this mechanism Moostrap supports behavioral reflection, i.e. a reification message passing.

6.2 Operating systems

6.2.1 Apertos

Apertos [Yokote 93],[Horie et al. 95] is an object-oriented operating system based on the principle of reflection. Every concurrent object has an own execution environment, called metaspace. The metaspace is, for example, responsible for scheduling of the concurrent objects in their object group. Objects can migrate to a different metaspace. Objects can explicitly interact with their metaspace. There are metaobjects with a dedicated function. Reflectors are entry points to metaspaces. Base objects send a message to the reflector when communicating with a meta space. MetaCore is the micro kernel located at each node. The Context object represents a processor. A Context can be suspended and resumed. Contexts can be connected to an interrupt. If the interrupt is raised, the related Context is resumed. An object group consists of objects that belong to the same metaspace. An object can be member of more than one object groups.

6.2.2 Choices

Choices [Madany et al. 92] provides reflective additions to C++. Classes are first-class objects. Choices relies on a preprocessor, several code manipulation tools and programmer conventions.

Choices allows the reification of:

- storage
- object existence (garbage collection)
- persistence (placing objects at garbage collected stable storage)
- class and inheritance (classes as objects, support of dynamic code linking, loading classes by name)
- encapsulation (using MMU hardware to support encapsulation of objects - ObjectProxy)
- inspection (symbolic object names, information about objects)

6.2.3 OMOS

OMOS [Orr 95] is not an operating system but provides a service that is often included in the operating system - the loading and linking of an application. OMOS is a load-time MOP, although it also can change module configurations at run time. OMOS has a configuration language that allows flexible combination of module files, so called dot-o-files. The configuration contains information how to map procedure names to procedures. It is possible to use inheritance at the level of dot-o-files.

7. Related concepts

7.1 Open implementations

Open implementations [Kiczales 96] are a recent area of interest in operating system research, intended to solve a problem called mapping dilemma. A mapping dilemma arises in black box systems if the black box is used in a non-anticipated manner.

Open implementation research has suggested a dual-interface approach, one interface providing the functionality and one adjustment interface. [Maeda 96] uses the open implementation approach to make the cache management strategy of a UNIX file system user customizable. Additional to the traditional base interface (open, close, read, write) he introduces a second interface that can be used to implement an application specific cache management strategy.

7.2 Aspect-oriented programming

[Kiczales et al. 96] realized that most complicated programs consists of different intervened aspects. They suggest to create one program for every single aspect in a language best suited for that particular aspect.

To build a program from the different aspects an *Aspect Weaver* is used. [Kiczales et al. 96] does not mention how the Aspect Weaver works and how the connection between the aspects is established while the aspects are separated.

7.3 Adaptive programming

The assumption of adaptive programming (AP) [Lieberherr 95] is, that most algorithms are spred over several classes and depend on the global class structure. Adaptive programming tries to separate algorithms from a specific class structure. In an adaptive program algorithms do not use method calls at objects from different classes, instead they use *propagation patterns* to traverse the object graph. Propagation patterns are separated from the class structure. If the class structure evolves, the algorithm (propagation pattern) must not be changed. The adaptive programming group (Demeter project) developed an extension to the C++ language called AP/S++.

7.4 Composition filters

Composition filters [Bergmans 94] are a mechanism to flexibly modify message sends and receives in the class-based language Sina. Currently there exist the following filters: Dispatch, Meta, RealTime, Error, Wait. Filters are specified in the interface part of a class definition. This makes it necessary to change the interface definition if the class is used with a different filter. The filter mechanism is similar to MetaJava's reification of incoming and outgoing messages. However, MetaJava provides no explicit support for filtering messages.

Chapter II
Design and Implementation of MetaJava

1 Reflection in Java

Java has already some reflective facilities built in. These include

- Classes and interfaces are first-class objects of type Class. This enables inspection of classes. A class object contains the classname, the superclass, the classloader, a list of interfaces implemented by the class. The class of every object can be determined with a call to this.getClass().
- The ClassLoader class enables loading of user generated classes.
- Finalization informs the object about its own deletion and enables the object to free all resources it holds.

Sun recently released JDK 1.1. This version of Java contains a "Reflection API", that enables structural reflection at run time. Unfortunately, there is no causal connection between the internal and the external self representation.

2 Structure of a Java virtual machine

This section describes an abstract model of the Java virtual machine (JVM). An understanding of the structure of the JVM is necessary to understand the modifications needed for the MetaJava system.

Figure II.1 displays the important components of a JVM and the relations between them, following the description in the JVM specification ([Lindholm & Yellin 96] pp. 57-82). The JVM model is intended to be as implementation neutral as possible. It contains all necessary parts of a Java system:

- Bytecode interpreter (cf. Section 2.1).
- Object store and garbage collector (section Section 2.2).
- A method area containing the data of classes and methods. The class store (section Section 2.3) is part of the method area.
- Java stacks. The operand and variable stack can be combined, because their size is known in advance.
- A collection of native functions.

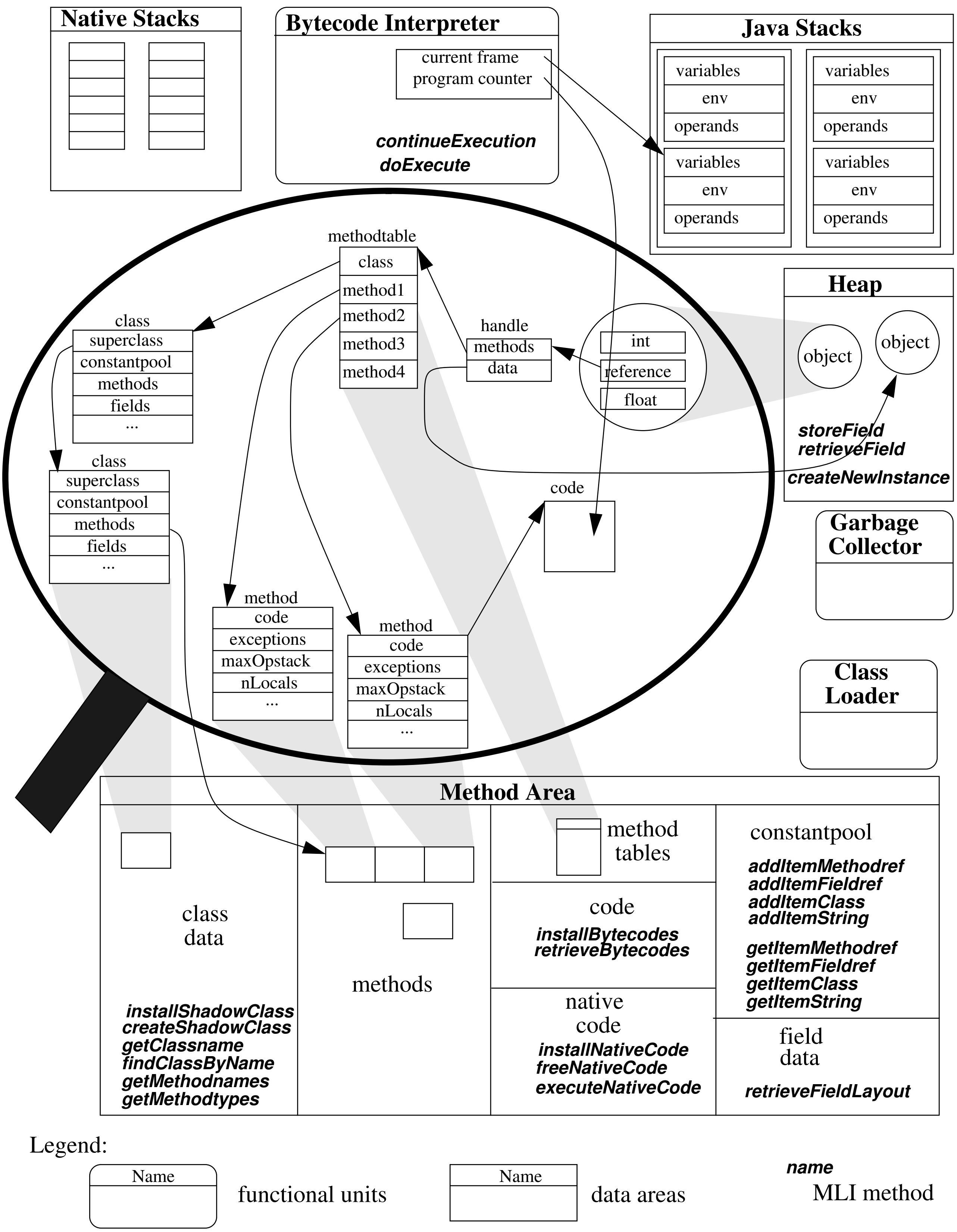


Figure II.1 Abstract model of a JVM

The specification of the VM [Lindholm & Yellin 96] enforces a certain structure of the VM implementation, however, the designer of a VM has freedom in a lot of points. Some of the alternatives are discussed in the following sections.

2.1 Bytecode interpreter

The interpreter main loop reads has two registers: a program counter and a stack pointer. It also has a reference to the current constant pool. The interpreter reads the bytecode at the current program counter position and executes code that implements this bytecode (cf. Figure II.2).

```
do {
    fetch opcode at current program counter
    fetch operands
    execute opcode
}
```

Figure II.2 Fetch-execute loop of the bytecode interpreter

We focus on the implementation of two bytecodes which have high importance for the meta architecture of MetaJava, these are invokevirtual and getfield.

The invokevirtual bytecode has one 16 bit argument consisting of two bytes in network byte order. This argument is a index into the constant pool of the method, containing an entry of type similar. The constant pool of a method is similar to a symbol table in languages like C or C++. All entries of the constant pool are typed. Available types are Utf8, Integer, Float, Long, Double, Class, String, Fieldref, Methodref, InterfaceMethodref, NameAndType. Some types are simple types (Utf8, Integer, Float, Long, Double), containing information directly available at the constantpool entry. Other types are compound types (Class, String, Fieldref, Methodref, InterfaceMethodref, NameAndType), referencing one or more other constant pool entries. A Methodref, for example, is a compound of a Class and NameAndType. A Class is a reference to Utf8 containing the name of a class. Utf8 is a simple type containing a unicode representation of strings.

The symbols can be resolved when the class is loaded (eager) or when they are first referenced by a bytecode (lazy) [1]. Implementations are likely to use the lazy approach, otherwise start-up time of applications would be unacceptable long. After a Methodref is resolved it points to a method block, a resolved Class points to a class block. In Sun's JDK implementation the invokevirtual bytecode is replaced by an invokevirtual_quick bytecode after resolving bytecode arguments. The arguments of the invokevirtual_quick bytecode are methodtable index (1 byte) and number of arguments (1 byte). If the number of methods or the number of arguments exceeds the range of one byte, a different quick opcode is used.

1. Implementations of C++ resolve method references at compile time, which avoids runtime overhead but leads to the fragile superclass problem.

During the start of a method execution a new stack frame is allocated. A stack frame consists of three blocks: a block to store local variables, an operand stack and an execution environment. The execution environment must contain the following information:

- A reference to the previous stack frame. When the method returns, either normally with a return bytecode or abnormally during an exception, the current stack frame must be released and the previous stack frame becomes the current frame.
- A program counter. The program counter points to the current opcode in the methods code block.
- A reference to the current class. This is used to verify accesses to other classes.
- Constant pool currently in use. This is a reference to the constant pool of the class the currently executing method belongs to. It is used to resolve symbolic references.
- Current monitor and time of monitor entry. The monitor is set when a synchronized method is entered. When exiting the stack frame, the monitor is released.
- Currently executing method.

The execution environment is best placed between the variable and operand block. The variable stack of the current frame should overlap the operand block of the previous frame, because method arguments are pushed on the operand stack and are expected at the local variable stack.

2.2 Object store and garbage collector

There exist some design alternatives to build an object store. We will discuss some of them an their possible implication for the reflective mechanisms of the system.

Direct vs. indirect object references

An object store can be build with direct references to the objects or with one indirection level, sometimes called object handles. The choice of one of these models has consequences for the applicable garbage collection algorithms. If a copying garbage collection scheme is used, direct references have to be patched if the object is moved. This reference updating requires the garbage collector to place a forwarding pointer at the old location of the object. While the reference is traversed, it is changed to the forwarding pointer.Figure II.1 displays an object store with indirect references (handles).

Object identity

Java objects have a unique identity. The identity of two objects is determined with “==” at the language level and opc_if_acmp[eq,ne,lt,gt,le,ge] at the bytecode level. These bytecodes can be implemented by bitwise comparing references.

Mark & Sweep

A lot of GC algorithms are described in the literature [Wilson 92]. In practice the most commonly used is the mark and sweep algorithm. In a first phase it marks all reachable objects starting from a root set following the references in the reachable objects. In the second phase, all unmarked objects are deleted. Before deletion of the object, the finalization method of this object is called. The root set of a garbage collector in a Java system consists of

- The Java stack and native stack of every thread.
- The constant pool of all loaded classes. The constantpool may contain `String` objects that used by the `ldc` opcode.
- Static variables in all loaded classes.

2.3 Class store

A *class block* contains all necessary information about one class. This information is extracted from the corresponding class file. As described in Section 1 some information contained in the classblock can be accessed through the `Class` object. A class block contains the following information:

- The class name.
- A reference to the superclass.
- A reference to the Java object for this class.
- A reference to the class loader used to load this class.
- The constantpool used by this class.
- All methods defined in this class.
- All fields defined by this class. Instance variables have only a type entry here. Static variables can store their value directly in the field description.
- A list of interfaces implemented by this class.

This information is directly derived from the class file. Additionally, in most implementations a class block contains derived information, for example the method table for objects of this class.

A method is represented by a method block and contains the following information:

- The name of the method.
- The signature of the method.
- Access flags of the method. Access flags contain the protection of the method and additional attributes (e.g., static, nativ).
- A reference to the class this method belongs to.
- The bytecode of the method.

- A list of exceptions thrown by the method.
- Number of local variables of method.
- The maximal space on the operand stack consumed by this method.

According to [Lindholm & Yellin 96] classes and methods are part of the method area, which is not garbage collected. In the new version of Sun's interpreter classes and methods are garbage collected. To garbage collect classes the collector must follow the class link in the method table of all reachable objects.

2.4 Start-up and initialization

The start-up phase of a JVM consists of initializing all global system resources, for example the object store and thread manager, and loading some basic classes, like java.lang.Object and java.lang.Class. After the system is initialized, the class containing the static main method is loaded, prepared, and initialized [2]. A array of string containing the arguments is constructed and the main method is executed.

The structure of a Java interpreter should now be sufficiently clear. The knowledge about JVM internals is indispensable for an understanding of the changes performed by MetaJava to enable behavioral reflection.

2. The class constructor is executed.

3 Design of the meta architecture

This section explains how the design issues of reflective architectures, which were discussed in Chapter I.5, are tackled in MetaJava.

The central design principle of the meta-level architecture is:

A meta-level program performs only non-trivial, exceptional computation.

This principle is motivated by the assumption that most meta-level systems will implement services like persistence and replication and thus access slow hardware components, like disks and networks. The principle guides the design of MetaJava and can be derived to:

- It is more important to not affect the performance of the base-level system, than to make the transition between base and meta level efficient. Modifications made for meta-level operations should induce only a local performance penalty.
- The system must be implemented efficiently. The meta level should not impose performance overhead when it is not used.

Additionally to this performance related design principle, the meta architecture should have the following properties:

1. The separation of concerns must be visible from design to coding phase. Functional units which are identified as being separate in the *design* phase must also be *implemented* separately.
2. Meta-level programs must be reusable.
3. Base-level programs must be reusable.
4. The architecture must be general, i.e. several problems like persistence, distribution, replication, synchronization must be solvable in it.

3.1 Scope

MetaJava's behavioral reflection system should consist of all implementation independent JVM mechanisms that may be altered in a meaningful way. We have identified the following mechanisms as fulfilling this requirement:

- The mechanism of method invocation. A method invocation is the most characteristic behavior of object-oriented programs. The implementation of the method invocation mechanism at the meta level should extremely enhance the customizability of base-level programs.

- The access to an instance variable. Objects consist of methods and variables. Beneath the method invocation mechanism, the mechanism to read and write an instance variable is a central mechanism in Java programs.
- The access to locks. Every Java object can be used as a binary semaphore, or mutex lock. If the mechanism to acquire or release such an object lock can be reimplemented by a meta-level program, several policies to maintain the lock can be applied.
- The mapping from classnames to class blocks. To perform its task, an object creates other objects, invokes methods of other objects, and so on. Most of these activities involve classes. If a new object is created, or a method is invoked at an object, the class of this object must be available. Initially, an object only has the name of the class. If it wants to do useful work with it, the name must somehow be transformed to a reference to a real class. If a meta-level program can implement this transformation, it is possible to control the classes that are visible to a specific object.
- Loading classes. The previous point was concerned with the resolution of class names, which is an object local mechanism. The system also contains a central component that loads classes into the interpreter. If a meta-level program is able to implement this global mechanism, it should be possible to use different mechanism of class loading depending on the class name.
- Creation of new objects. After classes are available in the system, new instances can be created from those classes. Like class loading, instance creation is a global mechanism. If a meta-level program could control instance creation, it might be possible to change the internal representation of an instance.

Any of these mechanisms has a default behavior that is implemented in the virtual machine. It should be possible to activate this default behavior at any time if the context is sufficient.

A reification of these mechanisms aimed at providing behavioral reflection capabilities. If a meta-level program should implement these mechanisms, it needs information about the structure of the base-level program. It will be necessary, to support structural reflection.

3.2 Computational model

The aim of this section is to provide the programmer with a model of MetaJava that is simple and easy to understand.

3.2.1 Separation in base level and meta level

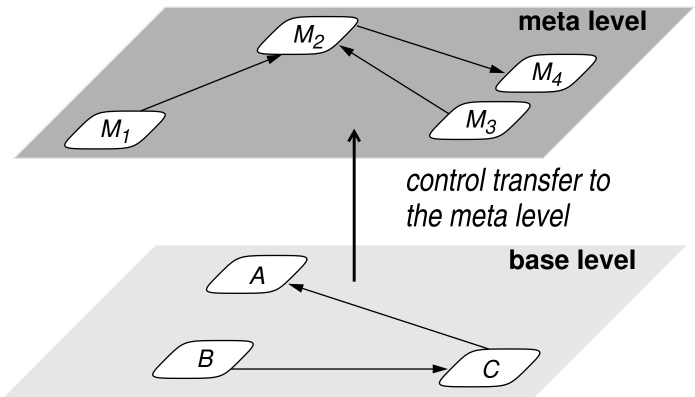
It was implicitely assumed in the previous section that computation in MetaJava takes place at a base level and a meta level. We will now explain this separation in detail. Base-level computation aims at solving a dedicated application problem, for example computing taxes in a financial program. Meta-level computations are all computations that do not directly solve the application problem, but compute about the base-level computations. The meta level of the financial application could, for example, move tax declarations between different hosts.

3.2.2 Control transfer

A fundamental issue of reflective architectures is the mechanism of transferring control from the base level to the meta level. There are some properties the control transfer mechanism should have:

- The control transfer should be transparent to the base-level object. The programmer should not need to consider any meta-level functionality when designing and implementing the base-level object.
- The control transfer should be synchronous. This gives the meta level a chance to control concurrent executions of base-level mechanisms. Different synchronization mechanisms can be implemented as meta-level programs.

MetaJava uses synchronous event delivery to meet these requirements. Figure II.3 shows the two levels of a program. Base-level computations use one of the mechanisms described in Section 3.1. If a metaobject implements this particular mechanism, an event is raised and control is transfered from the base level to the meta level.

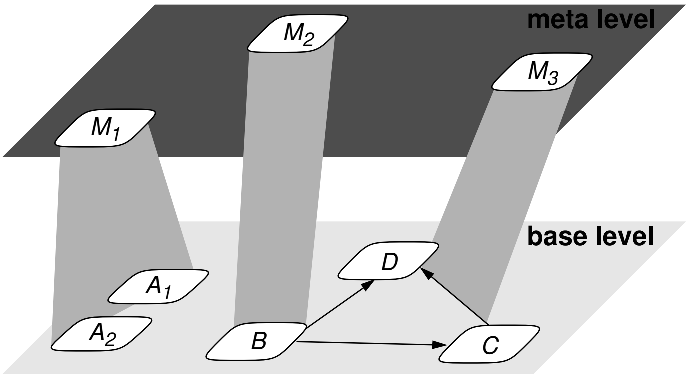


The computations of objects A,B,C raise an event, which transfers control to the meta level.

Figure II.3 Transparent control transfer to the meta system

3.2.3 Attaching metaobjects to base-level entities

Metaobjects can be attached to objects, classes, and references (see Figure II.4).



Metaobject M_1 is attached to class A. Objects A_1 and A_2 are instances of class A. Metaobject M_2 is attached to object B. Metaobject M_3 is attached to one reference from object C to object D.

Figure II.4 Binding of base and meta level

Attaching a metaobject to a base-level entity means that all events involving this base-level entity are delivered to the attached metaobject. If a metaobject is attached to a class, all events generated by instances of the class are delivered to the metaobject. If a metaobject is attached to an object, all events generated by this object are delivered to the metaobject. If an object is attached to a reference, only the events generated by operations using this reference are delivered to the metaobject.

An example should illustrate the differences. *A* metaobject may reify method calls. If this metaobject is attached to an object (like M_2 is attached to *B* in Figure II.4), all method invocations at this object are reified by the metaobject. If a metaobject is attached to a reference, all method invocations that use this reference are reified by the metaobject. In Figure II.4 invocations of methods of object *D* are only reified if they are performed by object *C* using the reference controlled by M_3. If *B* invokes a method of *D*, no reification takes place.

A metaobject has three methods to attach itself to a base-level object:

`void attachObject(Object obj)`

Attach metaobject to object.

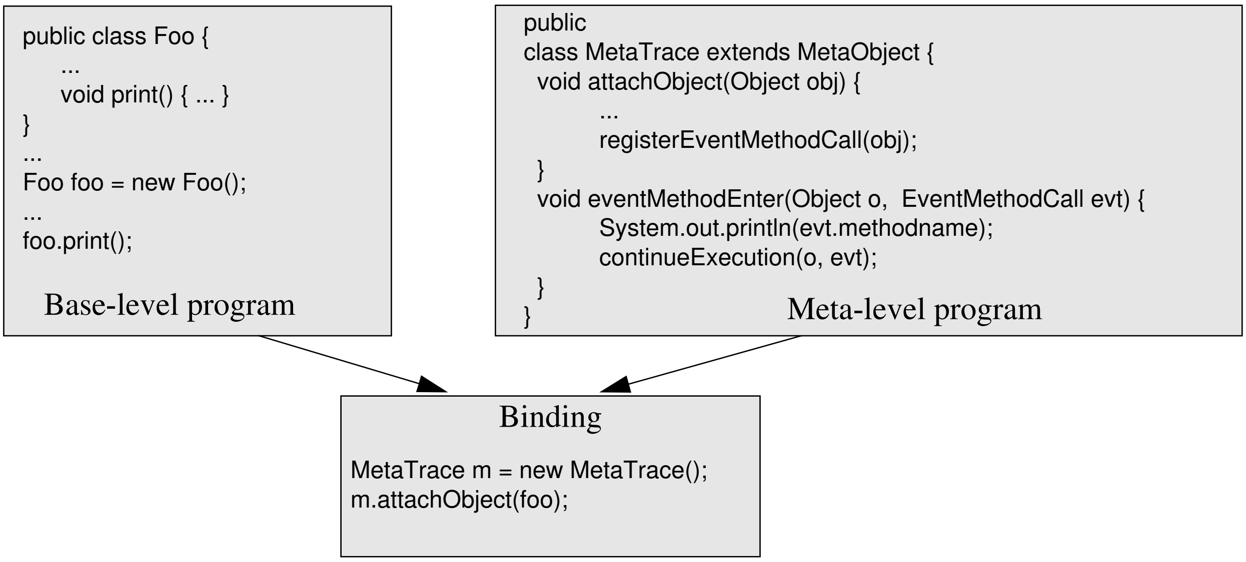
`Object attachReference(Object obj)`

Attach metaobject to object reference. It is not possible to *change* the semantics of a reference. The method returns a *new* (reflective) reference.

`void attachClass(Class c)`

Attach metaobject to class.

As Figure II.5 shows, the binding between base level and meta level can be done without modifications of code of the involved base-level and meta-level objects.



This system consists of three parts: a base-level program, a meta-level program, and a code sequence to bind base level and meta level. The binding code invokes the attach method of the metaobject with a reference to the base object.

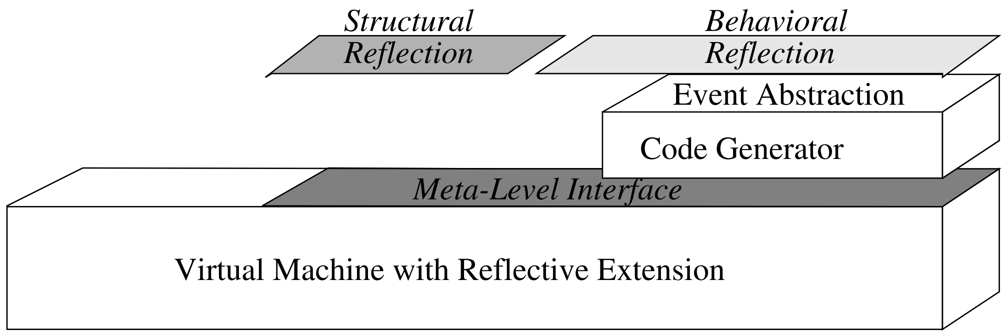
Figure II.5 Example binding of base and meta level

However, binding base-level object and metaobject seems to violate transparency. But transparency is only a local property. The example of transparency in a network file system (NFS) may illustrate this statement. The location of files in NFS is transparent to the user. But at one time this location transparency must be abandoned - when mounting the file system. Our binding procedure could be compared with the mount procedure of NFS.

Binding to objects should be done immediately after the object has been created. Binding to classes should be done at application start-up or before initial use of the class. However, the current implementation does not enforce these conventions. But note that it usually is not easy to understand the effect of changing the semantics of references, objects, and classes on the fly.

3.3 Structure

The MetaJava extension consists of two layers. The bottom layer opens the JVM and is called the meta-level interface. The top layer is written in Java and provides the described event mechanisms (cf. Section 3.2.2).



The MetaJava system consists of two layers: the meta-level interface to the JVM and a layer that provides an event abstraction using run-time code generation.

Figure II.6 Structure of MetaJava

This separation has some advantages:

- It eases experimentation with new event mechanisms.
- Because no new bytecodes are introduced for event generation, the system could possibly be ported to Java processors.

3.4 Meta-level interface

All methods are sorted in groups with related functionality. The functions of the MLI are separated in:

Manipulation of object data. The purpose of this group is the modification of an object's state. The group contains functions to access the instance variables of objects. It also contains a function to retrieve the layout of objects, i.e. the names and types of all instance variables.

Code modification support. This group contains functions to retrieve the bytecode of a method and to install new byteocde. These functions are used by the code generator.

Binding of base level and meta level. This group contains the low level interface for binding base-level and meta-level. Metaobjects use this functions to implement the attachment.

Modification of class structures. Metaobjects need to change the semantics of single objects or references. They perform these changes by modifying class structures.

Modification of constant pool. The constant pool of a class must be changed if methods of the class are changed.

Native code support. Contains functions to install and execute native code. Native code can directly be executed by the processor.

Execution. This group contains several functions that enable the execution of arbitrary methods.

Instance creation. These functions are related to instance creation.

A complete listing and description of the methods of the MLI is contained in Appendix A.

3.5 Behavioral reflection

As described in Section 3.2.2 behavioral reflection is achieved using event generation. The event generation system is a layer above the MLI (cf. Section 3.3). It provides the abstraction of behavioral reflection. The event generation layer for local events needs no further extension of the JVM. It is implemented completely in terms of structural reflection facilities (cf. Section 3.7). If the metaobject is interested in a specific behavior of the base-level object, it registers the event with one of the following methods.

Reification of incoming messages

We consider some alternatives. The first one is a scheme similar to before/after methods of CLOS. It is also influenced by [Reitzner & Kleinöder 96]. The before/after notification scheme is illustrated in Figure II.7. A meta-level method is executed before the base-level method and a (possibly but not necessarily different) meta-level method is executed after the base-level method. As suggested in [Reitzner & Kleinöder 96], methods

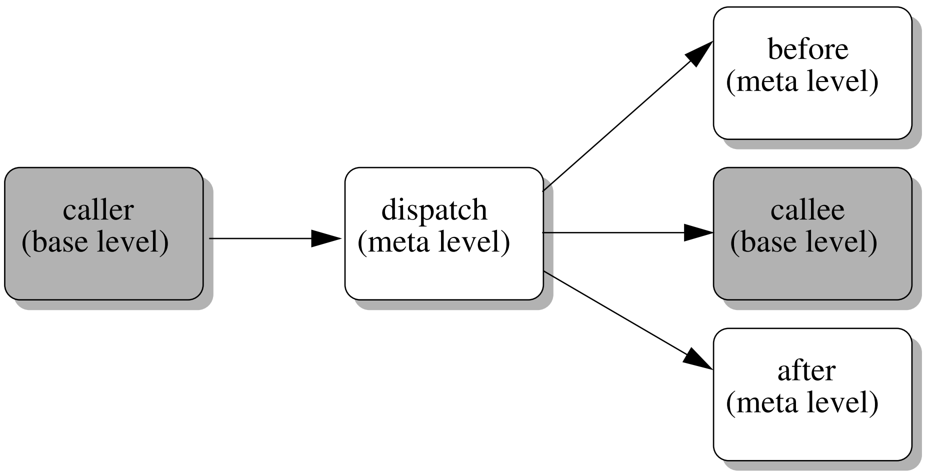


Figure II.7 Notification of method invocation (before,after)

have an attribute *methodclass,* additionally to their name and signature. This methodclass can be specified when attaching the metaobject at the base-level object. The before and after methods are selected according to the methodclass of the base-level method. The methodname of the responsible meta-level event handler is a concatenation of the methodclass and “Entry” for before method, or “Exit” for after method. The before/after handler expect no arguments.

The before/after notification scheme enables the individual metaobject to verify some constraints before or after executing a method or to delay method executions. But this scheme is not able to circumvent execution of the base-level method. When such a feature is needed, the handler based notification of Figure II.8 can be used.

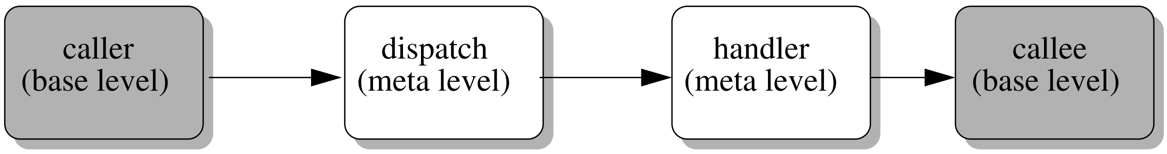


Figure II.8 Notification of method invocation (handler)

The handler is supplied with all information about the base-level method. Especially, the handler gets a list of all arguments. The handler is fully responsible for the execution of the base-level method. If the handler returns, the base system assumes that the method has been executed.

The presented two notification schemes are only examples. They are the only notification schemes currently implemented in MetaJava, but a meta-level program can implement new notification schemes if this is necessary for the application at hand.

To receive method invocation events the metaobject must register with the event handling layer using the method

`void registerEventMethodCall (Object obj, int notificationType, String methods[])`

> The object `obj` is the base-level object. The argument `notificationType` specifies the desired notification scheme. Currently only `methodNotificationHandler` and `methodNotificationBeforeAfter` are available. The array `methods[]` contains the concatenation of methodnames and signatures. Optionally, a methodclass can be specified in parenthesis. A method with name `test` and signature `(ILjava/lang/String;)V` must be specified with test `(ILjava/lang/String;)V`. If the method belongs to the methodclass `Reader`, the specification would be
> test `(ILjava/lang/String;)V(Reader)`.

Reification of outgoing messages

Method invocations performed by the base-level object at other objects are outgoing messages. To receive an event for an outgoing message, the metaobject must register with the event handling layer using

`void registerEventMethodCallOut (Object obj, String methods[])`

> The object `obj` is the base-level object. The array `methods[]` contains the concatenation of methodnames and signatures as described above.

Reification of variable accesses

Accesses to instance variables of base-level objects are notified similar to method executions. The metaobject can install a write handler and a read handler. The write handler is called when the variable is going to be written, the read handler is called when the variable is going to be read. The write handler is passed the intended new value as part of the notification message. The read handler must return the value which it attempts to have read. The metaobject must use

`void registerEventFieldAccess (Object obj, String fields[])`

The object obj is the base-level object. The array `fields` contains a concatenation of the names and types of instance variables. The type must be given in parenthesis. A field with name `city` and type `Ljava/lang/String;` is specified as `city(Ljava/lang/String;)`.

Reification of locking

To receive an event if the lock at a base-level object was acquired, the metaobject must use the method

`void registerEventObjectLock (Object obj)`

The object obj is the base-level object.

Reification of object creation

To receive an event if an object is created from class `clazz`, the metaobject must call

`void registerEventObjectCreation (Class clazz)`

Reification of class loading

To receive an event if the class with name `classname` is going to be loaded the metaobject must call

`void registerEventClassLoading (String classname)`

3.6 The causal connection requirement

As stated in Chapter I.5.1 the causal connection is a very strong requirement. For efficiency reasons, and unlike meta-circular interpreters, we use two separated self representations. One representation is internal to the JVM and designed for performance. The other representation is used at the meta-level by Java programs. The causal connection is realized with calls of the MLI that update the internal or external representation respectively.

3.7 Structural reflection

The MLI is not a comfortable interface for developers of meta-level programs. For this reason there exists a number of classes that make meta-level programming convenient. These classes allow for structural reflection of the internal organization of a base-level program.

MetaClass. Is the representation of a base-level class at the meta level. A MetaClass instance can be created be supplying an object, a class, or a name. The MetaClass instance can be used to list all methods of the class. The methods are represented as instances of class MetaMethod.

MetaMethod. The MetaMethod object provides a method to get the code of a method. The code is returned as an array of Opcode objects. MetaMethod also provides a method to inject opcodes into the method code. Jump addresses that cross the point of injection are corrected automatically.

MetaField. The MetaField object represents a field (instance or class variable).

Opcode. Opcode is the base class for all Java opcodes. Every Opcode provides a method to write itself into a byte array, which can then be installed as the code of a method with MetaObject.installMethodBytecode. MetaObject this low level procedure and provides a method to install an array of Opcode objects as method bytecodes.

4 Implementation

This section describes certain problems introduced by the addition of reflective capabilities to the JVM. It also describes how those problems are solved in MetaJava.

4.1 Shadow classes

To change the semantics of one object while not affecting semantics of other objects of the same class, the MetaJava system uses shadow classes. [Ferber 89] 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 [Ferber 89] seems only 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. MetaJava's shadow classes solve this problem.

Def. II.9 **Shadow class**

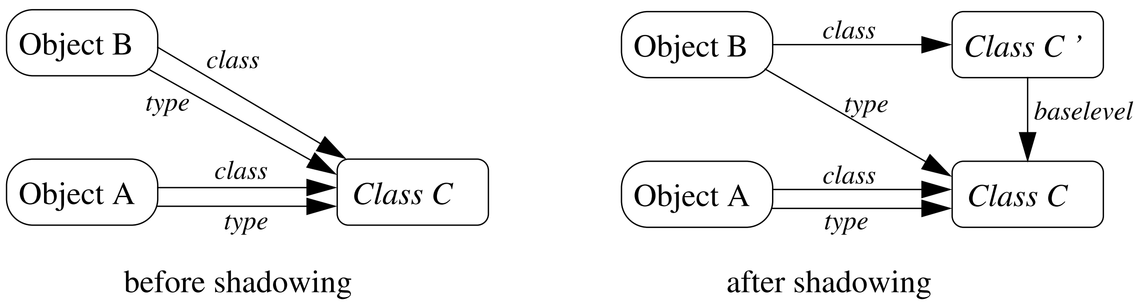
A shadow class C' is an exact copy of class C with the following properties:

– C and C' are un distinguishable 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 making the shadowing transparent to the base system.



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 II.10 The creation of a shadow class

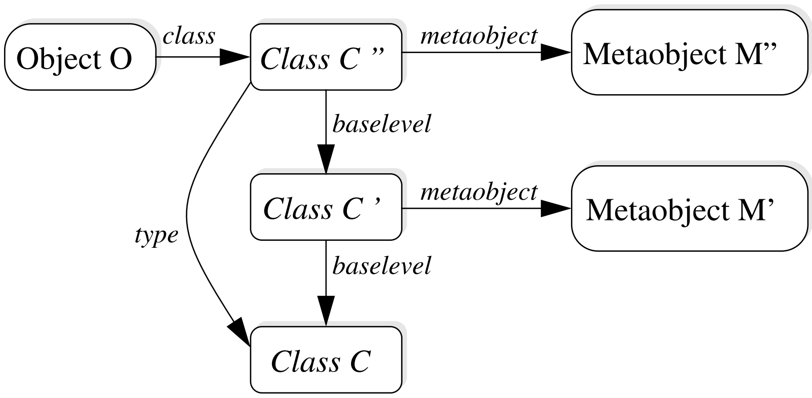
The type link is set automatically to the original class. We introduce shadow classes to solve a problem which is inherent in class-based languages. A class-based language assumes that there usually exist many objects with the same type (class). Prototype based languages like Self [Chambers 92] or Moostrap support one-of-a-kind objects [Mulet et al. 95]. 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.

The use of shadow classes introduces several problems:

- 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 should not change class related, but object related properties.
- Shadow classes must be garbage collected if they are no longer used, i.e. when the metaobject is detached.
- 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 implemented in MetaJava, affects the robustness of the system.

A shadow class C' is a shallow copy of C. Only methods 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)) byte per shadow class in MetaJava.

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* (cf. Figure II.11).



A metaobject M” is attached to an object that already has the metaobject M' attached. When attaching M” a shadow class C” is created. The type link of class C” is set to class C.

Figure II.11 The tower of metaobjects

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

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

constructed so that each metaobject reifies structure and behavior of the metaobject one level below. The MetaJava tower of metaobjects is similar to the composition filters model (cf. Chapter I.7.4) in that a metaobject can continue (delegate) work to the base level if it has finished its own computations.

The class located at the end of the baselevel chain is used for all type checks, e.g.

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

For fast access, the type link is also part of the class structure.

In Java classes are first class objects. Testing the class objects of C and C' for equality must yield true. In the MetaJava prototype every class structure contains a reference to the Java object of 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 JVM must compare the type links.

Because classes are first class objects it is possible to use them as mutex lock. This can happen 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 we use the class object of the class structure pointed to by the type link for locking.

Shadow classes and inheritance

We consider now how the shadow class mechanism blends with inheritance. During shadow class creation a shadow of the superclass is created recursively. When the object reverse to its superclass with a call to super the shadowed superclass is used. All shadowed classes in the inheritance path use the same metaobject.

Shadow classes and garbage collection

The garbage collector must follow 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.

4.2 Reification of virtual machine operations

This section describes the implementation of the behavioral reflection mechanisms introduced in Section 3.5.

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

4.2.1 Reification of incoming messages (method invocations)

To implement reification of method invocations we investigated the following alternatives:

- All interesting bytecodes (invokevirtual and invokespecial) are replaced by reifying pedants. This is a clean way to create reflective objects but requires an extension of the bytecode set.
- The interpretation process of interesting bytecodes is extended.
- Bytecodes are replaced with a stub code which jumps to the meta space. This is uncritical if the stubcode is no longer than the replaced bytecode. Otherwise method code must be relocated and extra care is needed with relative jumps across the stub code. This approach can be considered a special case of run time code generation.

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 chips. Therefore we decided to reify method invocations with stub generation at run time.

The next problem is how to make method arguments available to the metaobject. 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 MetaJava prototype arguments are passed in a special argument container which is able to change the type of one slot dynamically. This is necessary, because Java is a strongly typed language. The following primitive types are available: int, float, long, double, char, boolean, byte. Other types can created using classes and interfaces and inheritance. Primitive types are generally incompatible to each other (except the type conversion rules specified in [Lindholm & Yellin 96]). Non-primitive types are derived from class Object. This makes it necessary to “work around” the type system. A new class TypeAdaptiveElement is introduced which is able to store data of arbitrary type. Figure II.12 shows part of the interface of this class.

```
class TypeAdaptiveElement {
    String getElementSignature ( ) { ... }
    void storeInt (int el) { ... }
    int retrieveInt ( ) throws WrongTypeException { ... }
    void storeObject (Object el) { ... }
    Object retrieveObject ( ) throws WrongTypeException { ... }
    ...
}
```

Figure II.12 The class TypeAdaptiveElement

The recently stored type can be obtained with getElementSignature. The object. adapts to the type of the most recently stored element. If the client tries to retrieve a different type, a WrongTypeException is thrown.

A container of TypeAdaptiveElements can be build using an array, a Vector, etc. A more efficient implementation is provided by class TypeAdaptiveContainer using native methods. Figure II.13 shows part of the interface of this class.

```
class TypeAdaptiveContainer {
    String getContainerSignature ( ) { ... }
    void storeInt (int index, int el) { ... }
    int retrieveInt (int index) throws WrongTypeException { ... }
    void storeObject (int index, Object el) { ... }
    Object retrieveObject(int index) throws WrongTypeException{...}
    ...
}
```

Figure II.13 The class TypeAdaptiveContainer

4.2.2 Reification of outgoing messages

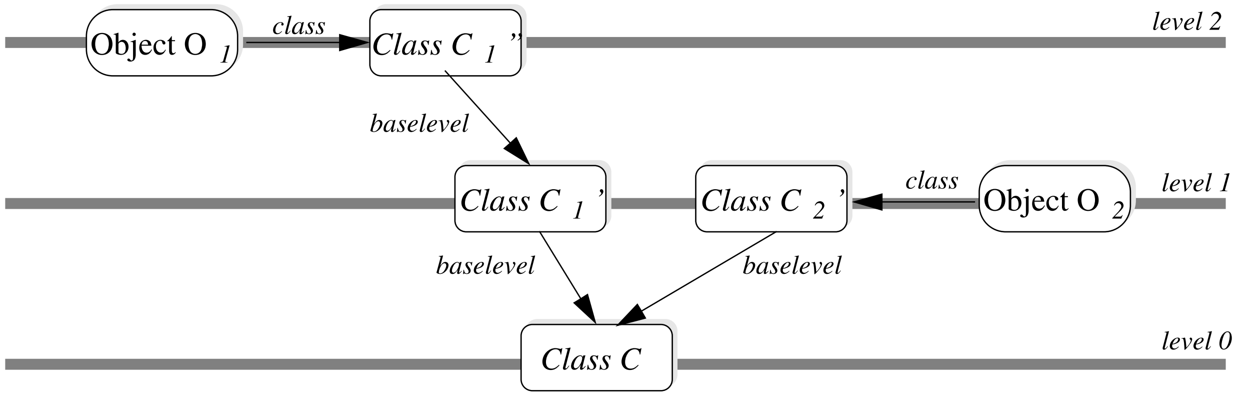
An outgoing messages is caused by the invokevirtual opcode [5]. The invokevirtual opcode has three arguments: a class name, a method name and a method signature. When registering for the event, the class name, method name, and signature can be specified. The code generator then looks for all invokevirtual 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

4.2.3 Reification of instance variable accesses

Reification of global accessible instance variables would either restrict the JVM 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 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. As illustrated in Figure II.14 the object could already be at a high meta level. If accesses to instance variables of O_1 should be reified, the putfield/getfield opcodes in all methods of C_1” must be replaced by a stub that calls the event handler of the attached metaobject. Unfortunately this is not enough to guarantee that all accesses to instance variables of O_1 are handled by the metaobject. Object O_2 can access private protected variables of O1 unnoticed by the metaobject of O_1. 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.



This figure shows that it is complicated to locate all code sequences that access an instance variable of O_1. The achieve this reification the code generator must find all putfield/getfield opcodes that access a variable with the according class/name/type. Object O_2 has the same class as O_1 and, hence, can access the variable of O_1.

Figure II.14 Problems with class based protection

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. Compare the top word on the stack with the this pointer for equality. This ensures that only accesses to object O are reified. If the comparison fails, the putfield is executed and method execution is continued normally.

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

3. Invoke the static eventDispatchVoid function of the class MetaObject with the event object as parameter.

Relevant getfield opcodes are replaced by a similar stub.

4.2.4 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 be 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. In MetaJava this function is extended for event generation. It 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. The meta system can modify 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. As stated in Chapter I.5.4 we consider reflective overlap a programming error, like unterminated recursion. The meta-level event handler for lock L should not use any method that acquires the lock L, because this would lead to reflective overlap.

4.2.5 Reification of class loading

Class loading is reified similar to object locking. If a metaobject is interested in the loading of a specific class, class loading is delegated to this metaobject.

4.2.6 Reification of object creation

The MLI function informOnObjectCreation creates an hook into the object store. The metaobject can use a non-reflective MLI method for object creation: createInstance.

4.3 Native code installation and execution

The MLI provides a group of methods to install native code. Some architectures need additional care, if data is installed as code. A processor with separate data- and instruction cache must probable flush the data cache and invalidates the instruction cache before executing the native code. Otherwise the instruction cache could read an old value from main memory or the cache. The native code interface is currently only available for MetaJava running on the i386, i486, or Pentium processor. The i386 and i486 have unified data and instruction caches. The first-level cache of the Pentium is separated in instruction and data cache. The instruction cache is read-only, because instructions fetched by the CPU are never written. However, the cache hardware detects self modifying code and invalidates the appropriate cache line [Intel 95]. Thus this function needs not worry about caches when using those processors.

RISC processors usually have an instruction pipeline. This pipeline must be flushed when native code is installed. Again the i486 and Pentium does this automatically.

The execution of native code usually involves the manipulation of object references. This could lead to problems with the garbage collector. If an object reference is only existent at the native code stack and not in any life object, the garbage collector would reclaim this object. Thus native code must follow certain rules when pushing references onto the stack.

4.4 Metaobject attachment

4.4.1 Attaching to classes

Additional to the entries listed in Section 2.3 every class structure contains an entry metaobject. This entry points to the metaobject that is bound to this class. When garbage collecting classes, this link must be used to mark the metaobject.

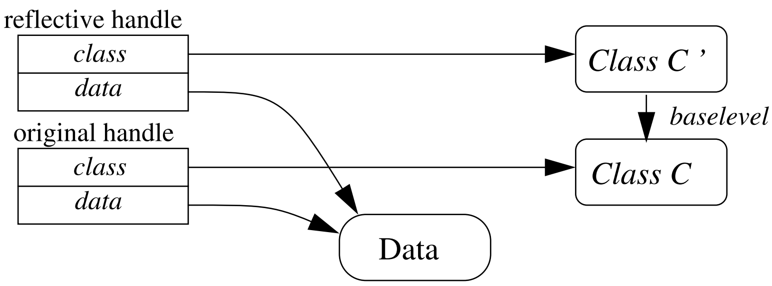
4.4.2 Attaching to objects

After attaching a metaobject to an object, the metaobject in most cases will change the semantics of the base-level object. This change is accomplished with a modification of the base-level object's class. Because a typical JVM implementation will share the class between all instantiated objects of this class, a shadow class must be created before modifying the 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.

4.4.3 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 (cf. Section 2.2). 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/

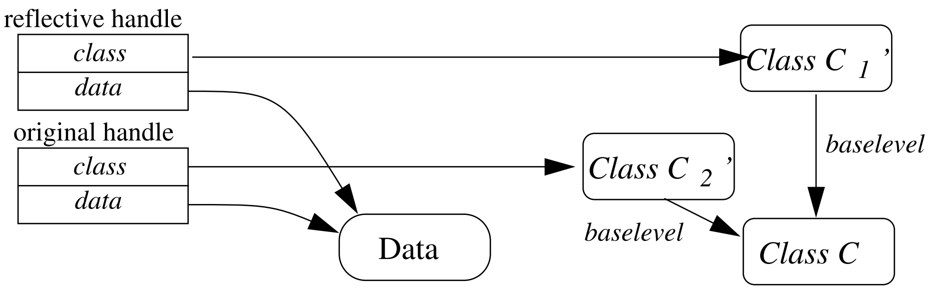
Figure II.15 Attaching to a reference

4.4.4 Multiple Binding

Section 4.1 explained that multiple metaobjects can be bound to one base-level entity (cf. Figure II.11). The most recent bound metaobject is activated at first. If this metaobject decides to continue with the default base-level action, the next lower 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 different types [6] of base-level entities. Currently, it is only possible to attach metaobjects in a certain order from more general to more specific entities. The most general one is the class, followed by the object and the reference. It is, for example, not recommended to attach a metaobject to an reference and then to a object. The MetaJava system allows this, but the semantics of an object attachment would be different from the specification [7].



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

Figure II.16 Attaching to a reference

When using multiple metaobjects with one base-level entity the programmer has to manually resolve any occurrences of semantic overlap (Chapter I.5.3).

4.4.5 Detaching metaobjects

A metaobject is detached from a base-level object by setting the `class` link of the object to the class pointed to by the `baselevel` link.

6. classes, objects, or references
7. The specification (Section 3.2.3) states that if a metaobject is attached to an object, the semantics of the object can be controlled by the metaobject.

4.5 How to find the metaobject

In most other object-oriented reflective languages the reference from the base object to the metaobject is part of the base objects state. In MetaJava the link from the base to the meta system is implemented using shadow classes. The shadow class contains a reference to the metaobject. An alternate option would be to include the metaobject link in the object or handle structure. Including it in the handle structure would increase the size of the handle space about 50%, because handles are small entities with the size of 2 pointers. The same problem exists with objects, because Java objects are usually small, containing no or only a small number of instance variables.

4.6 Performance

Performance should be a crucial concern in meta-level architectures. Unfortunately, no performance measurements are published for the majority of the reflective systems.

MetaJava tries to concentrate the overhead to those parts of the system, that do actually use the meta-level features.

Def. II.17 **Execution path**

An execution path is the sequence of instructions that are executed in order to provide a certain service.

The aim of the MetaJava system is, not to lengthen the execution path[8] of services not subject to reflection, thus making only local changes.

This aim is easily achieved when making method invocations reflective, because a method invocation activates code at a central location (the method to be made reflective). All mechanisms (see Section 3.1) that do not have this property create problems. A look at instance variable accesses illustrates the problem. The user intends to reify all accesses to a certain instance variable of an object. According to the Java scoping rules, the instance variable can be accessed from any point in the program, if it is declared public. Catching all accesses means modification of the implementation of the getfield bytecode in the interpreter main loop or modification of all code sequences possibly accessing the instance variable. Detecting the relevant sequences is not an easy task. An access to an instance variable is specified by giving a class name and a variable name and type. The accessed object is taken from the top of the stack. All sequences containing the name of the object's class or any superclasses and the variable name must be modified to transfer control to the meta level. This affects the execution of any access to these instance variable of every object of these classes. The first action is to check if the object on the stack is the same as the object under control of the metaobject. If not, execution continues with an ordinary getfield. If yes, control is transferred to the metaobject. To limit the extend of code modifications, public instance variables should be avoided. If the variable has

8. Sometimes also called critical path or code path.

private scope, only methods of the dynamic class of the object must be modified. No other classes are affected, and thus it is even possible to make a shadow class, reaching the highest degree of localization.

4.7 Metaobjects and garbage collection

Metaobjects are ordinary objects and thus subject to garbage collection. If the metaobject is not reachable from a root set, it is deleted. A metaobject should be deleted, if

- no reference to the metaobject is contained in live objects and
- and the metaobject is no longer attached to a base object

The first condition is assured by the unmodified garbage collector. The metaobject field of all classes must be added to the root set of the garbage collector.

5 Performance measurements

We tried to evaluate the impact of an empty meta level for the performance of several operations. Table II.18 lists the results. The column labeled “Java” displays the performance of a JVM with no meta architecture. The column labeled “MetaJava-noMeta” shows the slowdown of operations if the JVM contains all modifications necessary to use metaobjects. The column labeld “MetaJava-null meta” shows the performance degradation if a metaobject that implements the default semantics of the operation is attached to the base-level object. All numbers are in µs. The measures were taken at a slightly loaded SparcStation 20. All operations are excuted in a loop. The time needed for an empty loop was subtracted from the measured times.

Mechanism	Sun JDK 1.0.2	Java	MetaJava-no meta	MetaJava - null meta
empty loop	1.0	4.0	4.0	4.0
method call no arguments	2.1	6.1	6.1	170
method call 1 argument	2.3	6.6	6.6	230
method call 2 arguments	2.5	7.6	7.6	250
field read/write	1.7	6.0	6.0	140
acquire/release lock	15	22	28	350
object creation	10	12	20	170

Table II.18 Performance of operations in MetaJava (in µs)

These numbers are from an entirely unoptimized system (no * _quick optimization, etc.). They are expected to change if the MetaJava system is optimized. The slowdown of locking and object creation is caused by a violation of the design principle stated at the beginning of Section 3 A modification with global effect was made. But for now there seems to be no better way to reify locking and object creation.

6 Security

One of the keys of Java's success is the ability to interchange code in a computer network. This feature is, however, a major source of security problems. Java tackles the problems with a technique called sandboxes. All invocations of security relevant methods are checked by a security manager. All accesses of this code to local services is monitored and selectively granted. Importing meta-level code introduces new security risks, that, not surprisingly, can be handled using reflective techniques.

At start-up of an application a metaobject could be installed that controls all instantiations of metaobjects.

Chapter III
Programming at the Meta Level

1. Tracing method invocations: MetaTrace

The class MetaTrace, listed in Figure III.1, is a very simple meta object, which only prints out the method name and the arguments of the called method and then continues the base-level computation. Due to the different return types (void, int, Object, etc.) of Java methods, the event-handler methods have to be implemented for each return type.
The MetaTrace metaobject can be used to trace all method invocations at an object or only method invocations that use a certain object reference. If one wants to trace the methods invoked on a certain object, an instance of the MetaTrace class has to be attached to the object with

```
(new MetaTrace()).attachObject(obj, methods);
```

where obj is the base object and methods is an array of method names of the base object.

The tracing metaobject can also be attached to references with

```
new_ref = (new MetaTrace()).attachReference(ref, methods);
```

All method invocations at new_ref and references derived from new_ref are now traced by the MetaTrace object.

2. Remote method invocations: MetaRemote

Remote method invocations are implemented by the MetaRemote class and a class library that handles remote communication.

MetaRemote is a simple remote method invocation implementation. It does not use a nameserver for objectname resolution. Hence, it also does not use recursive reflection (cf. Chapter I.5.5).

If a node exports an object reference it must install an objectserver. The objectserver maintains a mapping from object IDs to local objects. Every node that intends to invoke a method at a remote object must install an object server for the method arguments, because arguments are passed by reference.

Figure III.2 shows what happens when an object invokes a method of a remote object. A program with similar behavior is given in Figure III.3. Prior to the method call, the calling object obtained a reference to the remote object in the form of a proxy. While, at first sight, this situation appears to be identical to ordinary remote method invocations, as in the Spring OS [Mitchell et al. 94] or in CORBA environments [OMG 95], there is an important difference: Normally proxy objects are used as stubs to forward a method call to its destination. In our case, the proxy is just an empty shell with a

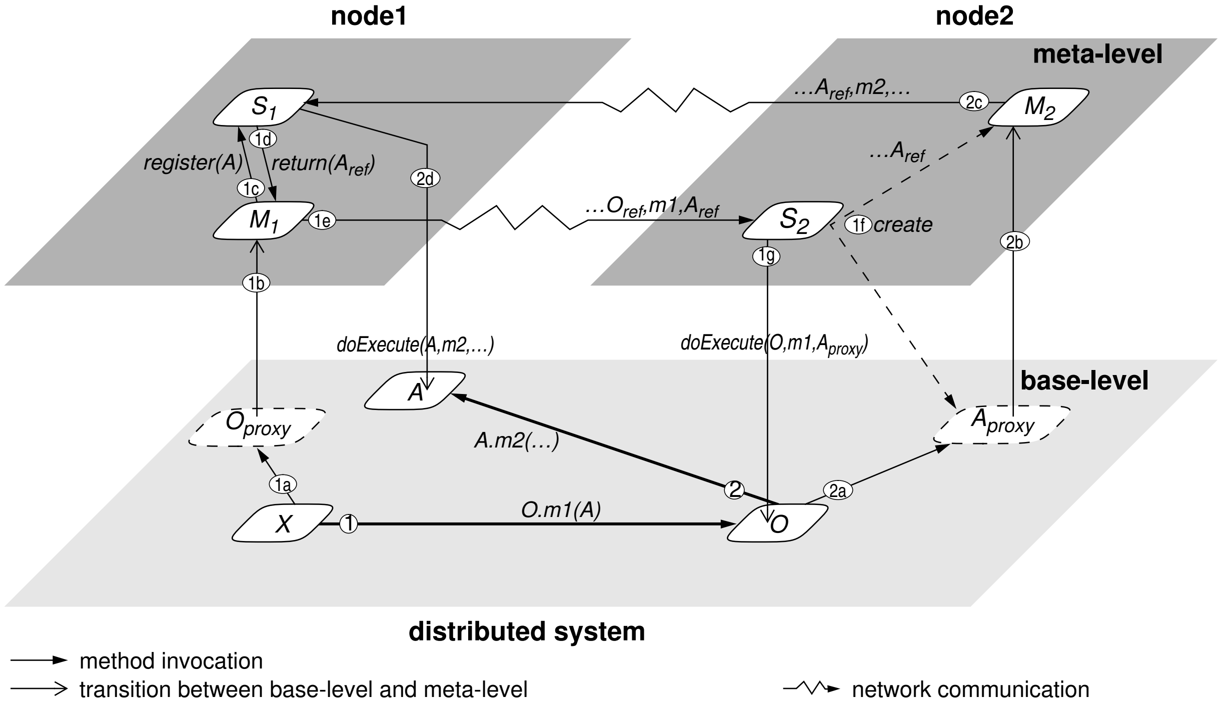
```
public class MetaTrace extends MetaObject
{
    public void attachObject(Object obj, String methods[] ) {
        Class shadow = MetaObject.createShadowClass(obj, this);
        installShadowClass(obj, shadow);
        registerEventMethodCall(obj, methodNotificationHandler, methods);
    }
    public void attachReference(Object obj, String methods[] ) { ... }

    public void eventMethodEnterVoid(EventMethodCall event) {
        trace(event);
        continueExecutionVoid( event);
    }
    public Object eventMethodEnterObject(EventMethodCall event) {
        trace(event);
        Object ret = continueExecutionObject(o, event);
        System.out.println("**   Returns object: "+ret.toString());
        return ret;
    }
    public int eventMethodEnterInt(EventMethodCall event) {  ... }
    ...
    private void trace(EventDescMethodCall event) {
        int type = TypeAdaptiveContainer.sig_Undef;
        System.out.println("** TRACE: Method " + event.methodname + " called!");
        System.out.println("**          Signature " + event.signature);
        if (event.arguments!=null) {
            for(int i=0; i< event.arguments.numberOfElements(); i++) {
            String sig=null;
            try {
                type = event.arguments.getElementSignature(i);
            } catch (UndefTypeException e) { ... }
            try {
                switch(type) {
                    case TypeAdaptiveContainer.sig_Integer:
                        System.out.println("**   Arg int: "+event.arguments.retrieveInt(i));
                        break;
                    case TypeAdaptiveContainer.sig_Object:
                        System.out.println("**   Arg object: "+event.arguments.retrieveObject(i).toString());
                        break;
                }
            } catch (WrongTypeException e) { ... }
        }
    }
}
```

Figure III.1 The implementation of the tracing meta object

metaobject attached to it. The proxy is only used to pass control to the meta level where the communication protocol is implemented (e. g. an interface to a CORBA ORB).

This way, catching remote method invocations is absolutely identical to method-call tracing or to the processing of access-control lists before allowing a method execution.

Garbage collection. The objectserver must maintain a reference count for each exported object. The metaobject at client side does not hold a reference to the proxy object. If the proxy is not longer used in base-level computation, it will be deleted. The metaobject must be informed about the deletion because it must request the remote objectserver to



1 Object X calls method *m1* of remote object *O*. This base-level action is implemented at the meta level as follows: 1a As *O* lives on a different node, *X* actually calls O_{proxy} . 1b Meta object M_1 is attached to O_{proxy} and thus it receives the *method-enter*-event. 1c M_1 registers the argument object *A* with object objectserver S_1 and 1d obtains the location-independent handle A_{ref}. 1e It calls objectserver S_2 with this handle, the method name, and the handle O_{ref} for the base object *O*. 1f Objectserver S_2 installs the proxy A_{proxy} for the argument object at node 2 and attaches meta object M_2 to it. 1g Objectserver S_2 calls the base-level object *O* using the meta-interface method doExecute( ). *O* executes the method code (base-level computation) and 2 invokes method *m2* of argument object *A* (and reaches A_{proxy} 2a). 2b This call is caught by meta object M_2 and 2c delegated to S_1 at node 1. S_1 maps A_{ref} to the registered local reference *A* and 2d jumps back to the base-level using doExecute( ).

Figure III.2 Invocation of remote methods

decrement the reference count for the exported object. The finalize method of the local proxy can be used to trigger this procedure.

Performance tuning. Special meta-level implementations are outside the focus of this thesis, so we will just mention some examples of improvements to the meta system described. One possibility would be to copy at least immutable data types, such as String to the remote node instead of passing references. Copying mutable objects would require a coherence protocol between the replicas. A meta object can be used to guarantee read-only access to copied objects. An alternative — which, however, entails different semantics — would be to copy them without coherence guarantees as in CORBA (with non-CORBA objects) or Sun's Java RMI [Wollrath et al. 96]. Hints for the optimal strategy for parameter passing (shallow or deep copy) could be provided by the base level with explicit meta interaction [Lopes 96]. Long proxy chains could be shortened on return of the remote method call, especially if the chain starts and ends on the same node. The current location of the remote object could be piggybacked with the reply. If objects are primarily used at a different node, it would be reasonable to let them migrate to that node. This could be done with a MetaMigrate[1] meta object.

```
import meta.remote.*;

class Arg {
    int v;
    public void set5() { v = 5; }
    public int get() { return v; }
}
class Callee {
    public void test(Arg a) {
        System.out.println("Test.test1");
        a.set5();
    }
}
class Caller {
    public void doit(String host) {
        MetaRemote m = new MetaRemote(host, 5555);
        Callee s = (Callee) m.createNewInstance( "meta/remote/Callee");
        Arg a = new Arg();
        s.test(a);
        System.out.println("called a.get(): " + a.get());
    }
}
```

This figure shows how the MetaRemote metaobject can be used. With createNewInstance a new instance of the specified class is created at the remote host and a local reference to a proxy of this object. is returned. All method invocations at this proxy are reified by m and forwarded to the remote host.

Figure III.3 Application of the MetaRemote meta

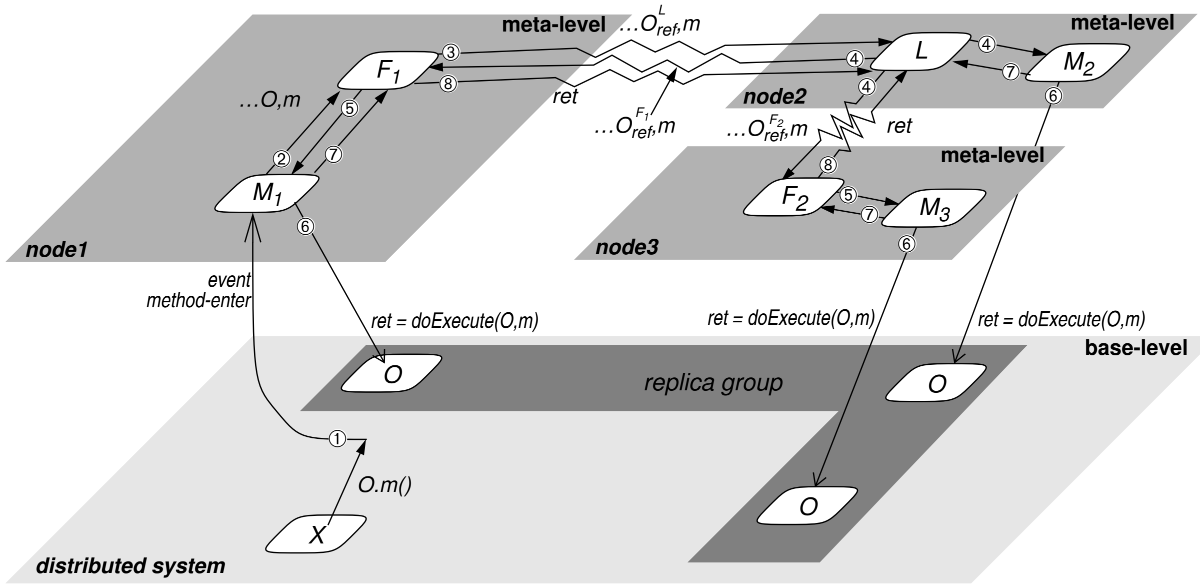
3. Replication: MetaReplication

The previous section described how the invocation of a method of a remote object is implemented at the meta level. We now describe the implementation of a meta-level system that replicates objects at different hosts and guarantees a certain consistency of the replicas. We have implemented two replication protocols, one based on active replication and one based on passive replication. We use the term *active replication* for every protocol that duplicates base-level threads between nodes. Every protocol that maintains a replicated state is classified as *passive replication*. Thus active replication protocols must reify method executions while passive replication protocols must reify instance variable accesses.

3.1 Object locking

Every Java object can be viewed as a binary semaphore. Duplicating an object allows distributed synchronization using the object as a distributed lock. The replication-con-

1. Not implemented yet.



method invocation
transition between base level and meta level
network communication

Object *X* calls method *m* of the replicated object *O*. This base-level action is implemented at the meta level as follows: ① Meta object *M* is attached to *O* and thus it receives the *method-enter* event. ② *M* informs the local ReplicationController about the method invocation attempt. ③ The ReplicationController happens to be a Follower. It sends a request to the Leader to execute the method. The request contains the object ID O^L_{ref} and the methodname and signature. ④ On receiving the request the Leader creates a new thread to distribute the method invocation to all followers and the local replica. It waits for the results from all followers and the local metaobject. ⑤ The followers inform the metaobject to execute the method. ⑥ The metaobjects $M_{1..3}$ execute method *m* at the base-level object *O* using the MLI method doExecute( ). ⑦ The return value of method *m* is given back to the ReplicationControllers $F_{1..2}$ and *L*. ⑧ $F_{1..2}$ send their result to *L*.

Figure III.4 Active replication: Method invocation at replicated object

trol meta object detects the attempt to acquire the lock with the MetaJava event mechanism. Every replication protocol must implement a mechanism to maintain the lock.

3.2 Active replication

A scenario of active replication is sketched in Figure 4. It shows All method invocations at O are reified by O's *replication metaobject* M. The *meta-level objects* $R_{1..3}$ together with M are responsible for a consistent execution of the replicated methods. $R_{1..3}$ implement the interface *ReplicationControler.*

We used a centralized protocol to implement lock management and ordered multicast [2]. The central component is the *leader* and the other participating nodes are the *followers* . Leader and follower are meta-level objects and implement the interface ReplicationController. A replicated object has a copy on every node. The copies are called replicas and all replicas form a replica group. Every replica has an attached metaobject M.

2. Our intention was *not* to compare different protocols and find heuristics how to select the optimal replication protocol for a specific base-level behavior. We only wanted to demonstrate the benefits of base/meta separation when implementing replication.

Figure III.4 shows the involved objects at the base level and the meta level and their interaction when a method at a replicated object is invoked.

When an execution (or lock/unlock) request arrives at the Leader from a Follower, the Leader executes the following operations:

1. send execution (lock/unlock) request to all Followers
2. tell the local metaobject to execute the method (lock / unlock the object)
3. wait for results from the Followers and the local metaobject
4. reply to the initiating Follower that the operation was performed

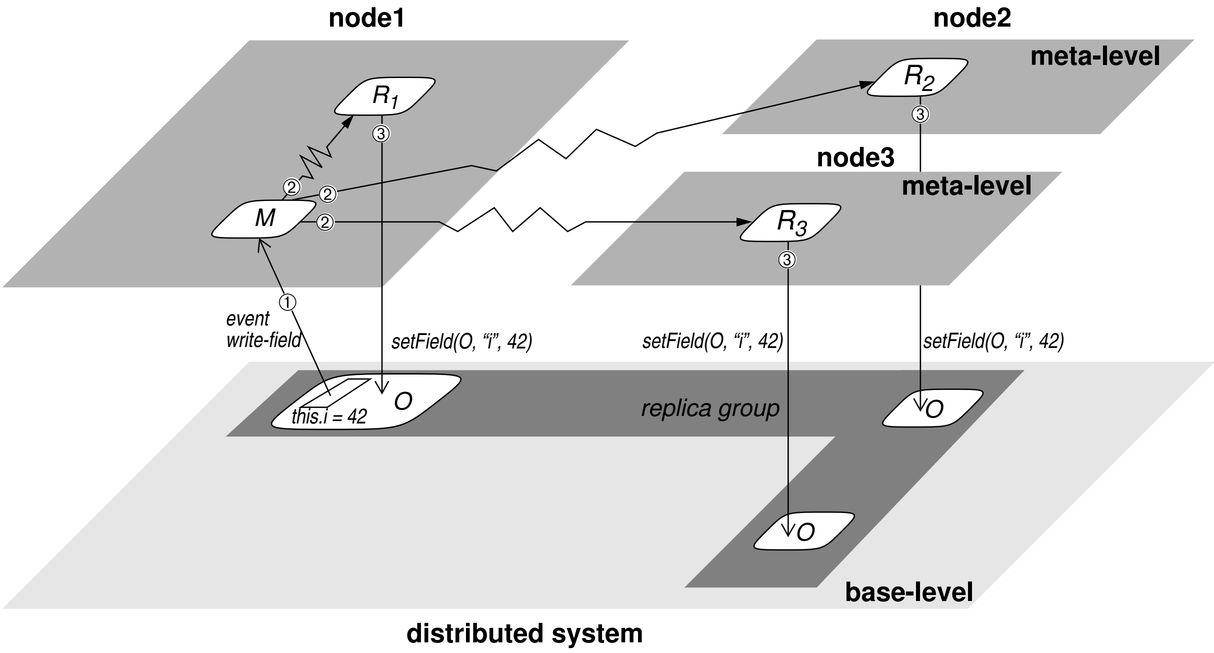
When an execution (or lock/unlock) request arrives at the Follower from a Leader, the Follower executes the following operations:

1. send execution (lock/unlock) request to the local metaobject
2. reply to the Leader that the operation was performed

When an execution (or lock/unlock) request arrives at a metaobject, the metaobject executes the following operations:

1. execute base-level method with `doExecute` (lock/unlock base-level object)
2. reply to the Leader that the operation was performed

If every method invocation is distributed to all replicas, one can hardly expect any performance benefits from replicating the object. Thus the replication meta objects use the information about the semantics of the method, which was supplied by the programmer when binding the replication-control meta object to the base object. If a method has no side effects and is read-only, the method is executed only at the local replica. All other method invocations and lock operations are performed by the whole replica group.



A method of object *O* is executed and tries to write field (instance variable) *i* . This base-level action is implemented at the meta level as follows: ① Meta object *M* is attached to *O* and thus it receives the *write-field*-event. ② *M* requests all replication-protocol meta objects of the replica group to set field *i* of object *O* to the new value. ③ $R_{1..3}$ set the field to the new value using the meta-interface method setField( ).

Figure III.5 Passive replication: Write access at replicated object

3.3 Passive replication

A possible implementation of passive replication is shown in Figure III.5. The replication-control meta object *M* reifies the access to all *fields* (instance variables) of the replicated base-level object *O*. The replication-protocol meta objects $R_{1..3}$ are responsible for a consistent state of the replicated object.

We use the term *replication meta objects* for all replication-control and replication-protocol meta objects together.

Our passive replication protocol ensures sequential consistency with a write-invalidate protocol. We implemented a slightly modified version of the write-invalidate protocol used in IVY [Tan et al. 90]. In IVY the unit of sharing is one page and the consistency-preserving protocol works on whole pages. In our implementation the unit of sharing is one object and the consistency-preserving protocol operates on fields. The replication meta objects keep track of the current state of each field of each replica (*current owner, writable, readable, invalid*).

If a field in one replica is to be written, control is passed to the meta level with the write-field event. If the field is not marked as writable, the current owner is requested to transfer the ownership to the writer. After receiving the ownership the field in all other replicas is invalidated. Finally the field is marked writable and the meta object sets the new value.

Read operations cause a read-field event. If the field is marked invalid, the owner is requested to grant read access. The owner marks the field in its own replica as readable, transfers the value to the requesting meta object and grants the read rights. The meta object marks the field as readable and returns the value as result of the read operation.

4. Just-in-time compilation: MetaJIT

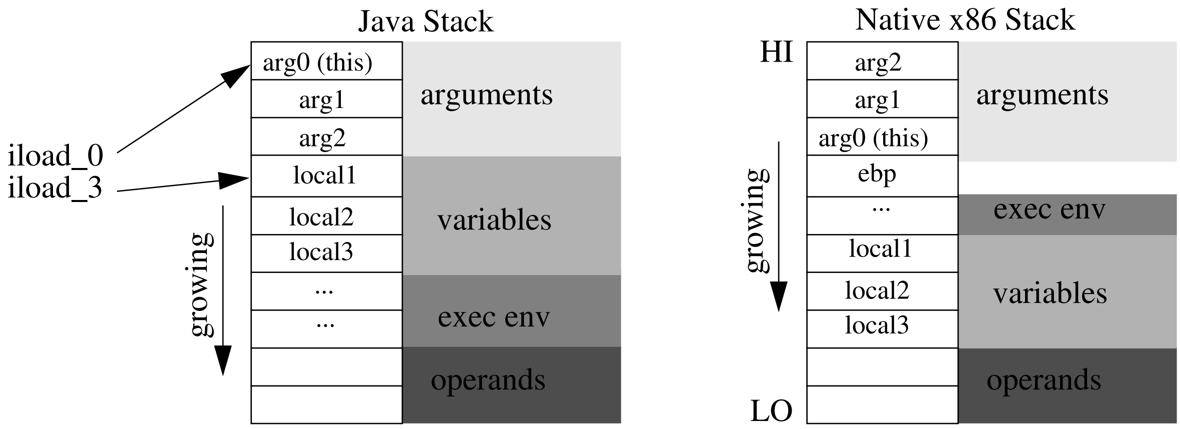
A promising way to improve the performance of a Java system is just-in-time compilation. Sun specifies an interface to plug a user-supplied compiler into the virtual machine [Yellin 96]. The MetaJava meta-level interface is a better way to install user-supplied JIT compilers into the system. There are several reasons for this:

- A JIT compiler in Java space can profit from the comfortable Java development environment (garbage collector, type-save language, etc.).
- It is possible to customize the compiler for specific application domains. The compiler could be used for numeric applications and extremely polymorphic programs.
- The compiler is only processor dependent, not platform dependent. It should be possible to use the JIT compiler for different JVM implementations that execute on the same processor architecture.

4.1 Code generation

The MetaJIT compiler generates non-optimized x86 code. Bytecodes are translated into native code using a direct mapping. A more elaborate design will use register allocation to enhance performance.

The native code stack is a combined variable and operand stack. The layout of Java and native stacks is compared in Figure III.6.



The left figure shows the layout of a Java stack that uses a contiguous area of memory for local variable block, execution environment, and operand block. The right figure shows the layout of a native x86 stack for the current MetaJIT implementation on a Unix system.

Figure III.6 Java and native stack

Currently, the execution environment is not used for native stacks. The following list explains with some examples the key problems of a JIT compiler.

- The compiler must save the stack pointer at method entry and restore it at method exit. The following code is generated for this purpose.

```
pushl ebp ; save ebp on the stack
movl esp, ebp ; save esp in ebp
subl esp, 4 * (n_locals - n_arguments) ; reserve space for remaining local variables

   ... ; main part of the method

movl ebp, esp
popl ebp
ret
```

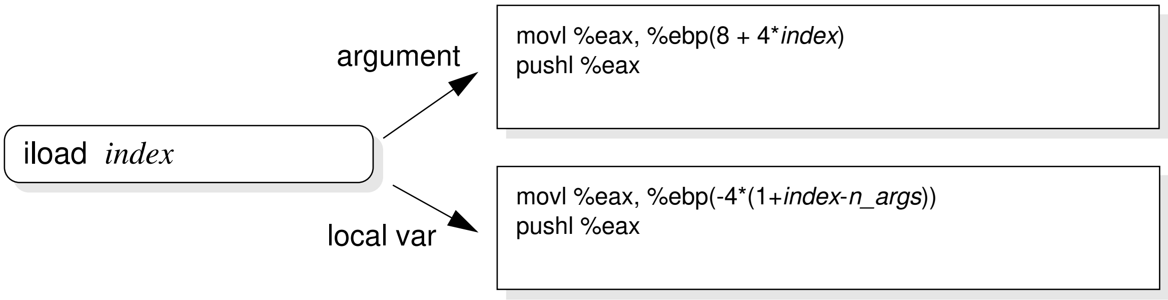
- Due to the similarity between Java and native stack the bytecodes for arithmetic operations can be compiled straightforward.

```
iadd
```

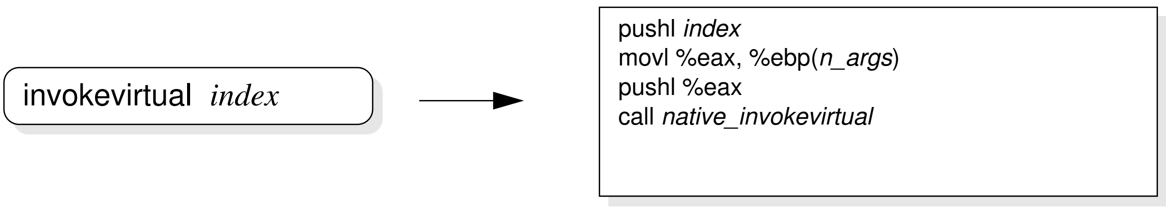
→

```
popl %eax
popl %ebx
addl %eax, %ebx
pushl %eax
```

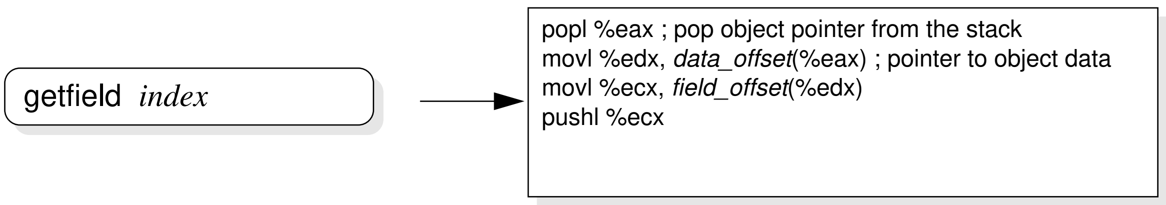
- A bit more complicated is the compilation of bytecodes that access local variables. The iload bytecode reads a local variable and push it on the operand stack. Different code must be generated if the local variable is an argument.



- A method can contain invocations of other methods. If the method is compiled into native code, the compiler must insert instructions to jump from native code back into the virtual machine. The MLI provides the address (VM-DataSheet.native_invokevirtual) of a function in the virtual machine. The JIT compiler can use this address to transfer control back to the interpreter. The function pointed to by *native_invokevirtual* expects arguments in Java layout.



- To compile the getfield opcode the JIT compiler must know, how to read a value from an object. This depends on the internal architecture of the object store. Some technique would be necessary to inform the compiler how to generate code that accesses an instance variable without exposing the JVM internals. The currently implemented JIT assumes an objectstore with indirect references as sketched in Figure II.1. *data_offset* is the offset of the data pointer in the handle structure. *field_offset* is the offset of the field in the data area of the object. *field_offset* is part of the MetaField information and can be retrieved with getOffset.



- To generate relative forward jumps the compiler must know how much code is located between the current position and the target position. This information is only available after the code has been generated. The compiler inserts the target after the code generation is completed.

if_icmplt →

```
popl %eax ; load value 2
popl %ebx ; load value 1
cmpl %ebx, %eax ; value1 - value2
jg target ; generated using a jump table
```

The garbage collector must scan references on the native code stack. This is no problem, because the GC must do the same think for functions that are implemented as native.

4.2 Status

The MetaJIT compiler has been used to compile some simple methods. Execution time of these compiled methods improved considerably. It can currently not be used to compile complex methods, because not all bytecodes can be compiled.

Chapter IV
Conclusion and Future Work

1. Conclusion

The goal of this thesis was to develop an extension to the Java virtual machine that enables a separation of programs into base-level and meta-level components. This goal was reached and it was demonstrated with a prototype implementation that it is feasable to extend the JVM for structural and behavioral reflection. The behavioral reflection system was based on an event model. The distribution of events is controled outside the virtual machine, so different event distribution schemes could be implemented. This thesis demonstrated the usefulness of behavioral reflection with a remote method invocation system, a replication system and a just-in-time compiler. But while the goal of base/meta separation could be reached, the much more elaborate goal of a meta/meta separation must remain subject of future research.

2. Future work

This thesis has generated at least as much questions as answers. The ideas for future work are classified into three groups: extensions to the MetaJava Virtual Machine, concepts to structure the meta space, and the construction of a library of metaobjects.

2.1 Extensions to the MetaJava Virtual Machine

- Implement a reification of constant pool resolutions as described in Chapter II.3.1.
- Attach metaobjects to threads. Attaching a metaobject to a thread can be used to trace all locks aquired by a certain thread or to control access to native functions for untrusted threads.
- Consider reification of arrays. Reification of array access could, for example, allow a space efficient storage of sparse arrays without the base level noticing this.
- Consider reification of Java stacks.
- Complete the MetaJIT compiler and find out how the compiler can be made independent from the internals of a particular JVM.

2.2 New concepts

- Investigate meta-level composition and techniques to resolve conflicts caused by non-orthogonal metaobjects.
- Use a combined compile-time run-time reflective system to enhance expressability.
- A more tightly integration of the reflective JVM into the operating system could provide more flexibility in adapting operating system related mechanisms to application needs.
- Currently locks are the only resources that can be managed outside the virtual machine by a meta-level program. The processor is a resource that also should be maintained outside the virtual machine. This is currently not possible, because the MetaJava system uses the *pthreads* interface which does not allow to modify scheduling decisions on a fine granularity.

2.3 Library of metaobjects

Migration control. Different strategies to control the migration of objects, e.g. the computational field model [Tokoro 90], can be implemented at the meta level as shown in [Okamura & Ishikawa 94].

Object clustering. Grouping of objects can sometimes provide more advantages than keeping objects and their memory location orthogonal [Cooper & Wise 95]. A meta object may control all object creations performed by a root object and its descendants, and place them into the same memory segment. With the separation of base-level and meta-level code, the grouping in segments can even be transparent to the base level, while the current location of the objects is visible at the meta level.

Security. Meta objects can be used to implement various security policies. For example, a meta object can be attached to a reference before passing it to an unsure object, to control access to it and its propagation. Such a meta object may be used for expiration or revocation of access rights to the reference. Meta objects can also serve as guards to check all accesses to a specific object. Special concepts, such as the Java security manager, can also be implemented by meta objects. Security meta objects can be very useful in Applet programming. Ideas how to use metaobjects to implement flexible security mechanisms are described in [Riechmann 96].

Active objects. Active objects are considered to be a suitable new programming paradigm for facilitating programming in multithreaded environments. Active objects have their own thread of control to execute their method code. Synchronization is done at the object border using message queues. We argue that active objects are just ordinary objects with a special meta object, called MetaActive, attached to them. MetaActive queues methods and executes them asynchronously in its own thread. The thread of the caller returns immediately. Returned references get a MetaFuture meta object, which suspends the execution of threads accessing the result until the method execution has been finished and the result has been computed.

Atomic objects. Atomic objects offer an interface with atomic guarantees: only when an operation is completed does it have an effect on the object state. Furthermore all changes are visible at once, usually when the method returns. Stroud and Wu [Stroud & Wu 95] use meta-object protocols to implement atomic data types with different concurrency-control algorithms. An implementation could create a shadow copy of the object's state and copy it back when the transaction commits.

Synchronization. Meta objects can be used to implement generic synchronization schemes — for example, constraining the sequence of events with path expressions, as proposed in [Campbell & Haberman 74]. Lopes and Liebeherr [Lopes & Lieberherr 94] separate synchronization patterns from functional code and use a special code generator to create an object-oriented program from a structural, a behavioral, and a concurrency code block. The concurrency code block could be implemented at the meta level. First ideas of implementing synchronization mechanisms using a model similar to MetaJava are presented in [Reitzner & Kleinöder 96].

Extended transactions. The classic ACID model of transactions is not sufficient for some application domains, especially for reactive applications [Eirich 94]. To make extended transaction models practical to use, Barga and Pu [Barga & Pu 96] propose a Reflective Transaction Framework which transparently assigns extended semantics to transactions.

Fault tolerance. [Fabre et al. 95] show how several fault-tolerance mechanisms can be implemented at the meta level.

Dependency management. A lot of code in object-oriented programs is concerned with managing data dependencies between several objects. The model-view-controller paradigm is a well known design pattern to implement this dependency management. [Ducasse et al. 95] discuss the advantages of making dependency management explicit.

3. The End

Dear Reader, you just reached the end of this thesis. I hope that the concepts of reflection and meta architectures are now clear for those who were unfamiliar with the topic. I am confident that the MetaJava system will prove as a valuable platform for future research in object-oriented reflective systems.

Bibliography

Barga & Pu 96

R. Barga and C. Pu. Reflection on a Legacy Transaction Processing Monitor. *Proceedings of Reflection '96*, San Francisco, Ca., April 1996. pp. 63–78.

Bergmans 94

L. Bergmans. *Composing Concurrent Objects*. PhD. Thesis. Universiteit Twente, Jun. 1994

Campbell & Haberman 74

R. H. Campbell and N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science,* vol. 16, Springer Verlag, New York, 1974, pp 89–102.

Chambers 92

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

Chiba & Masuda 93

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.

Chiba 96

S. Chiba. OpenC++ Programmer's Guide for Version 2. Technical Report SPL-96-024, Xerox PARC, 1996.

Cooper & Wise 95

T. Cooper and M. Wise. The Case for Segments. *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems,* Lund, Sweden, IEEE, 1995, pp. 94–102.

Chereque et al. 92

M. Chereque, D. Powell, P. Reynier, J.-L. Richier, J. Voiron. Active Replication in Delta-4. 22. *International Symposium on Fault-Tolerant Computing*, Boston, Ma., IEEE, 1992, pp. 28–37.

Danny & Malmkjær 88

O. Danny, K. Malmkjær. Intensions and Extensions in a Reflective Tower. *Conference Record of the 1988 ACM Symposium on LISP an Functional Programming,* 1988, pp. 327-341.

Dearle et al. 94

A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, F. Vaughan Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 7(3), Summer 1994. pp. 289-312.

Ducasse et al. 95

S. Ducasse, M. Blay-Fornarino, A.Pinna-Dery. A Reflective Model for First Class Dependencies. *Proc. of OOPSLA'95*. Oct. 1995. pp. 265-280.

Eirich 94

T. Eirich. *The ACID-Fission — Pay Only for What You Need*. Technical Report TRI4-09-94, University of Erlangen-Nürnberg, IMMD IV, Sept. 1994.

Ferber 89

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.

Fabre et al. 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.

Foderaro & Fateman 81

J.K. Foderaro, R.J. Fateman. Characterization of VAX Macsyma. *Proc. of the 1981 ACM Symp. on Symbolic and Algebraic Computation*, Berkeley, CA, 1981, pp. 14-19.

Friedman & Wand 84

D. P. Friedman, M. Wand. Reification: Reflection without metaphysics. Symposium on LISP and Functional Programming, Austin, TX, August 1984, pp. 348-355

Garbinato et al. 95

B. Garbinato, R. Guerraoui, K. Mazouni. Implementation of the GARF Replicated Objects Platform. *Distributed Systems Engineering Journal*, March 1995.

Goldberg & Robson 89

A. Goldberg, D. Robson. *Smalltalk-80*. Addison-Wesley 1989.

Horie et al. 95

M. Horie, E. G. Manning, G. C. Shoja. Exploring the Apertos Operating System. Technical Report of the Department of Computer Science, University of Victoria (DCS-238-IR), September, 1995

Hürsch & Lopez 95

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

Intel 95

Intel Corporation. Pentium(R) Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual, 1995

Johnson & Palaniappan 93

R. Johnson and M. Palaniappan. MetaFlex: A Flexible Metaclass Generator. *Proceedings of ECOOP '93, the 7th European Conference on Object-Oriented Programming,* Kaiserslautern, Germany, LNCS 707, Springer-Verlag, 1993, pp. 501–527.

Kiczales 96

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

Kiczales et al. 91

G. Kiczales, J.des Rivieres, D.G.Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

Kiczales et al. 96

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/).

Kleinöder 92
J. Kleinöder. *Objekt- und Speicherverwaltung in einer offenen, objektorientierten Betriebssystemarchitektur*. PhD. Thesis. July 1992. (in german)

Lieberherr 95
K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company.

Lindholm & Yellin 96
T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

Lindstrom 95
A. Lindstrom. Multiversioning and Logging in the Grasshopper Kernel Persistent Store. *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems,* Lund, Sweden, IEEE, 1995, pp. 14–23.

Lopes & Lieberherr 94
C. V. Lopes and K. J. Lieberherr. Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications. *Proceedings of ECOOP '94, the 8th European Conference on Object-Oriented Programming,* LNCS 821, Springer-Verlag, 1994, pp. 81–99.

Lopes 96
C. V. Lopes. Adaptive Parameter Passing. *2nd International Symposium on Object Technologies for Advanced Software,* Kanazawa, Japan, LNCS 1049, Springer-Verlag, March 1996.

Madany et al. 92
P. Madany, N. Islam, P. Kougiouris, R. H. Campbell. Reification and Reflection in C++: An Operating Systems Perspective. *Proceedings of the International Workshop on Reflection and MetaLevel Architecture.* Nov. 1992. pp. 76--82

Maeda 96
C. Maeda. A Metaobject Protocol For Accessing File Systems. *2nd International Symposium on Object Technologies for Advanced Software,* Kanazawa, Japan, LNCS 1049, Springer-Verlag, March 1996.

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

Maes 87b
P. Maes. Concepts and Experiments in Computational Reflection. *Proc. of OOPSLA '87*, Oct. 1987. pp. 147-155.

Masuhara et al. 92
H. Masuhara, S. Matsuoka, T. Watanabe, A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. *Proc. of OOPSLA '92*, 1992

McAffer 95
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.

McAffer 96
J. McAffer. Engineering the Meta Level. *Proceedings of Reflection '96,* San Francisco, CA, April 1996.

Mitchell et al. 94
J. Mitchell, J. Gibbons, G. Hamilton, et al. An Overview of the Spring System. *Proceedings of the COMPCON Spring 1994*, San Francisco, Ca., 1994.

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

Mulet & Cointe 93
P. Mulet, P. Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. *International Symposium on Object Technologies for Advanced Software,* Kanazawa, Japan, LNCS 742, Springer-Verlag, Nov. 1993.

Okamura et al. 92
H. Okamura, Y. Ishikawa, M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. pp. 36-47.

Okamura et al. 93
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.

Okamura & Ishikawa 94
H. Okamura and Y. Ishikawa. Object Location Control Using Meta-level Programming. *Proceedings of ECOOP '94, the 8th European Conference on Object-Oriented Programming,* LNCS 821, Springer-Verlag, 1994, pp. 299–319.

OMG 95
Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Rev. 2.0, July 1995.

Orr 95
D. B. Orr. Application of Meta-Protocols to Improve OS Services. *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. Orcas Island, Washington. May 1995. pp. 101-105.

Powell 91
D. Powell, editor. Delta-4 - A Generic Architecture for Dependable Distributed Computing. ESPRIT Research Reports. Springer Verlag, November 1991

Reitzner & Kleinöder 96
S. Reitzner, J. Kleinöder. *Synchronizing Classes: Splitting Inheritance Concepts*. Technical Report TR-I4-96-04, University of Erlangen-Nürnberg, IMMD IV, July 1996.

Riechmann 96
T. Riechmann. *Security in Large Distributed, Object-Oriented Systems.* Technical Report TRI4-02-96, University of Erlangen-Nürnberg, IMMD IV, Mai 1996.

Rodriguez 91
L. H. Rodriguez Jr. *Coarse Grained Parallelism using Metaobject Protocols*. Master's Thesis. Massachusetts Institute of Technology. September 1991.

Simmons et al. 92
J. Simmons, S. Jefferson, D. Friedman. *Language Extension via First-class Interpreters*. Trechnical Report 362, Indiana University, Computer Science Department, Sept. 1992.

Smith 82
B.C.Smith, *Reflection and Semantics in a Procedural Language*. PhD Thesis. Massachusetts Institute of Technology. MIT/LCS/TR-272. Jan. 1982.

Stroustrup 92
B. Stroustrup. Run Time Type Identification for C++. *USENIX C++ Conference proceeding*, Portland, Or., Aug. 1992.

Stroud & Wu 95

R. J. Stroud and Z. Wu. Using Metaobject Protocols to Implement Atomic Data Types. *Proceedings of ECOOP '95, the 9th European Conference on Object-Oriented Programming,* LNCS 952, Springer-Verlag, Aug. 1995, pp. 168–189.

Tan et al. 90

M. Tan, J. M. Smith, D. J. Farber. A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems. *Operating Systems Review*, Vol. 24, No. 3, Jul. 1990, pp. 40-67.

Tokoro 90

M. Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Computing Systems. *Proceedings of the 2nd Workshop on Future Trends in Distributed Computing Systems*, Cairo, Sep. 1990.

Voss 93

R. Voss. Time Invariant Member Function Dispatching For C++ Evolvable Classes. OOPSLA '93 *Workshop on Object-Oriented reflection and Metalevel Architectures,* Washington D. C., Oct. 1993.

Wilson 92

P. R. Wilson. Uniprocessor Garbage Collection Techniques. *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 1992.

Wollrath et al. 96

A. Wollrath, R. Riggs, J. Waldo. A Distributed Object Model for the Java System. *Proceedings of the Conference on Object-Oriented Technologies and Systems, COOTS '96*, Toronto, Jun. 1996, pp. 219– 231.

Yellin 96

F. Yellin. *The Java Native Code API*. Sun Microsystems, Inc. July 30, 1996

Yokote 93

Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. Position Paper for the *Workshop on Reflection and Meta-level Architectures* at OOPSLA 93.

Appendix A
Meta-Level Interface of the JVM

Manipulation of object data

Object internals must be accessed in an implementation independent way.

String retrieveFieldLayout(Object o)

Get layout of all fields of object o (object signature). This includes all fields declared in superclasses. The signature is constructed starting with the base class. The object signature also contains all static variables.

Type[1] retrieveField*Type*(Object o, int index)

Get contents of field with specified index. Indexing is according to signature returned by retrieveFieldLayout(). The requested field must be of type *Type* .

void storeField*Type*(Object o, int index, *Type* contents)

Store contents as new value of field. The requested field must be of type *Type* .

Code modification support

byte [] retrieveMethodBytecodes(Object o, String methodname, String signature)

Reification of method code. Returns the bytecodes of the specified method. The JVM specification describes a performance optimization, the *_quick* opcodes. The are used in Sun's JDK implementation in place of the original opcodes if the arguments containing indexes into the constant pool, are resolved. The byteocde reifier reverses this process.

void installMethodBytecodes(Object o, String name, String signature, byte code[], int maxstack, int locals)

Installs code as new (or modified) method. The maxstack parameter gives the maximal size of the operand stack needed by the new method. The locals parameter gives the number of local variables used in the new method.

Binding of base- and meta level

void attachObject (MetaObject meta, Object base)

Bind a meta object to a base object.

1. *Type* is a wildcard for the types: Object, int, float, boolean, byte, long, double, char

MetaObject findMeta (Object base)

Find the responsible meta object for a base object.

Manipulation of class structures

Class createShadowClass (Object base)

Create a shadow class of class `base`.

void installShadowClass (Object base, Class shadowClass)

Install a shadow class at base object.

Object cloneReference (Object ref)

Create a new reference which points to the same object as the given one. This method is used to attach meta objects to references.

Manipulation of the constant pool

int addItemString(Class c, String str)

Store string in constant pool of class c. Returns index to constant pool item.

int addItemClass(Class c, String classname)

Find class with name classname and store reference to found class in constant pool of class c. Returns index to constant pool item.

int addItemMethodref(Class cl, String classname, String methodname, String typename)

Find method and store reference to found methodblock in constant pool of class `cl`. Returns index to constant pool item.

int addItemFieldref(Class cl, String classname, String name, String type)

Find field block with fieldname `name` and fieldtype `type` and store reference to found fieldblock in constant pool of class `cl`. Returns index to constant pool item.

FieldDesc getItemFieldref(Class cl, int index)

Return field block at specified constant pool index.

FieldDesc getItemMethodref(Class cl, int index)

Return field block at specified constant pool index.

Class getItemClass(Class cl, int index)

Return class at specified constant pool index.

String getItemString(Class cl, int index)

Return string at specified constant pool index.

Native code support

int installNativeBuffer(byte code[])

This function allocates an executable memory buffer (“text segment”) and copies code into this buffer.

The native code buffers are accessible with a handle, which is returned by this function.

int allocNativeBuffer(int size)

This function allocates an executable buffer and returns a handle to the buffer.

void fillNativeBuffer(int handle, byte code[])

copy code into native buffer

int getBaseAddress(int handle)

get base address of native code buffer

VType[2] executeNativeBuffer*VType*(int handle)

execute native code, with no return value

void freeNativeBuffer(int handle)

frees resources associated with native code handle

VMDataSheet getDataSheet()

Get description of internals of the virtual machine implementation. Some meta-level programs need this information. The just-in-time compiler, for instance, must known the internal organization of objects to be able to generate efficient code.

Execution

VType continueExecution*VType* (EventMethodCall event)

Continue the execution of a method. This calls the non-reflective method. No event is generated, otherwise the reflection would not terminate.

VType doExecute*VType* (EventMethodCall event)

Execute a method. Contrary to the previous two methods, this one calls the method as if it would be called by an ordinary base object.

Instance creation

void informOnObjectCreation(MetaObject meta, Class clazz);

2. *VType* is a wildcard for the types: void, Object, int, float, boolean, byte, long, double, char

Register object creation. If an object is created from class clazz, the event handler of meta is called.

Object createInstance (Class class)

Create a new instance of a class.

Object locking

void informOnObjectCreation(Object obj;

Register object locking. If an lock on object obj is acquired or released, the event handler of the attached meta object is called.

void acquireObjectLock(Object obj)

Acquire the lock on object obj. Blocks if lock is hold by another thread.

void releaseObjectLock (Object obj)

Releases lock on object obj.

Class loading

void informOnClassLoad(String classname);

Register class loading. If the class with name classname is loaded, the event handler of meta is called.

Class loadClass(String classname)

Load class with name classname using the internal class loading mechanism.

Class installClass(byte classbuffer[])

Installs new class. The byte array classbuffer must contain the class data in a valid class file format.

Appendix B
Event Types

This appendix contains a list of all currently defined MetaJava events and their fields.

EventDescMethodCall

String methodname
String signature
String methodclass
TypeAdaptiveContainer arguments

EventDescFieldAccess

String fieldname
String signature
boolean reading
Type value*Type*

EventDescObjectLock

boolean acquire

EventDescClassLoading

String classname

EventDescObjectCreation

Class clazz

Appendix C
Classes for Structural Reflection

MetaClass

MetaMethod getMethod(String name, String signature)
int numberOfMethods()
Class getClassObject()
String getName()
int numberOfFields()
MetaField getField(int i)
MetaField getField(String name, String type)

MetaMethod

String getClassname()
String getName()
String getType()
Opcode[] getOpcodes()
int numberOfArguments()
void replace(Opcode code[], int mstack, int nlocal)
int replaceInject(int pos, Opcode code[], int add_maxstack)
void reflect()
String info()

MetaField

String getClassname()
String getName()
String getType()
int getOffset()

Opcode

int getPos()
int setPos(int i)
void translate(Class clazz, CodeBuffer c)
int len()
String dump()

OpcodeJump

void setTarget(Opcode t)

List of Definitions

List of Figures