MetaJava: An Efficient Run-Time Meta Architecture for Java™

Jürgen Kleinöder, Michael Golm

June 1996 TR-I4-96-03

Technical Report

Computer Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University Erlangen-Nürnberg, Germany

MetaJava: An Efficient Run-Time Meta Architecture for Java™

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

Abstract

Adaptability to special requirements of applications is a crucial concern of modern operating system[2] architectures. Reflection and meta objects are means to achieve this adaptability. This paper reports on ideas and experience we obtained while extending the run-time system of the object-oriented language Java with reflective capabilities.

We explain our model of an object-oriented architecture that allows flexible and selective attachment of reflective properties to objects. We show how reflection can be obtained with minimal changes to the existing system and how the penalty in run-time performance can be minimized. Our architecture is not limited to special application domains like distributed or concurrent computing but can also be used to support different security policies, just–in-time compilation, location control of mobile objects, etc. As an example, a remote method invocation mechanism is described to demonstrate how the Java programming model can be enhanced using our meta architecture.

1 Introduction

Today's applications demand more flexible support from operating systems (OS) for a variety of tasks and run-time properties, including distribution, security, persistence, fault tolerance, and synchronization. The OS should provide these services in a transparent way, so that application programs do not need to be modified if new or different run-time properties are required. One approach to providing this support is to extend the OS for the required services as was done for fault tolerance in Delta [7] or for persistence in the Grasshopper Kernel [15]. This strategy has some deficiencies. It is neither user customizable nor extendible. A second approach is to provide the required functionality in the form of libraries. This approach has the advantage of being user customizable but it is not transparent to the functional code of the application program.

We advocate a reflective software architecture to overcome these deficits while retaining customizability *and* transparency.

We use Java as our target system and call the extended system MetaJava. Nevertheless our principles may be used to equip other systems with reflective capabilities.

We are aware that we must impose some constraints to make our extension attractive to a large community of Java users. So we avoided changes to the compiler and the virtual machine. We merely added extensions to the virtual machine.

Section 2 introduces reflection and metaprogramming. Section 3 describes our computational model. Section 4 presents the design and implementation. Section 5 outlines some examples. We conclude with a comparison with related work and a short note about the current project status and future work in sections 6 and 7.

2 Reflection and metaprogramming

In the past programs had to fulfil a task in a limited computational domain. Demands of today's applications are becoming more complex: multithreading (synchronization, deadlock detection, etc.), distribution, fault tolerance, mobile objects, 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 features such as persistence. Reflection is a fundamental concept for a uniform implementation of all these different demands.

According to Maes [18], *reflection* is the capability of a computational system to "reason about and act upon itself" and adjust itself to changing conditions. *Metaprogramming* separates functional from non-functional code. *Functional code* is concerned with computations about the application's domain (*base level*), non-functional code resides at the *meta level*, supervising the execution of the functional

1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* Grant *Sonderforschungsbereich SFB 182*, Project *B2*.
2. When we talk about operating systems we include system libraries, run-time systems, and virtual machines.

code. To enable this supervision, some aspects of the base-level computation must be reified. *Reification* is the process of making something explicit that is normally not part of the language or programming model.

As pointed out in [9] there are two types of reflection: structural and behavioral reflection (in [9] termed computational reflection). Structural reflection reifies structural aspects of a program, such as inheritance and data types.
A common technique for structural reflection is to let one meta class control a number of base classes. Run-Time Type Identification (RTTI) of C++ [25] is an example of structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. Our architecture deals with behavioral reflection at run time. We show how a run-time meta architecture can be built that is nearly as efficient as one that operates only at compile time. A programming environment that incorporates a meta architecture gains the following advantages:

Increased productivity. Like object-oriented programming, metaprogramming is a new paradigm which leads to more structured and easily maintainable programs.

Separation of concerns. 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 [11], [14]. 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 together with the application program. We regard reflective decomposition as a new structuring technique in addition to functional and object-oriented decomposition.

Configurability. The meta level establishes an open system architecture [13]. New policies can be implemented without changing the application code. This is especially useful for class libraries, where the library designer can only guess the demands of the library user.

Not only application developers can profit from metaprogramming but also application users may replace meta components to tailor the application to their particular needs. Typically, a system administrator will tailor the meta space, so that the application can take care of local resources such as processors, printers, network connections, or hard disks.

Transparency. Transparency and orthogonality of base system and meta system are desirable features but cannot always be guaranteed in real applications. Consider a meta object that implements a bounded buffer synchronization scheme. It does not make sense to attach it to a base object that does not have a bounded buffer semantics (i.e., two operations *put* and *get* to access the buffer).

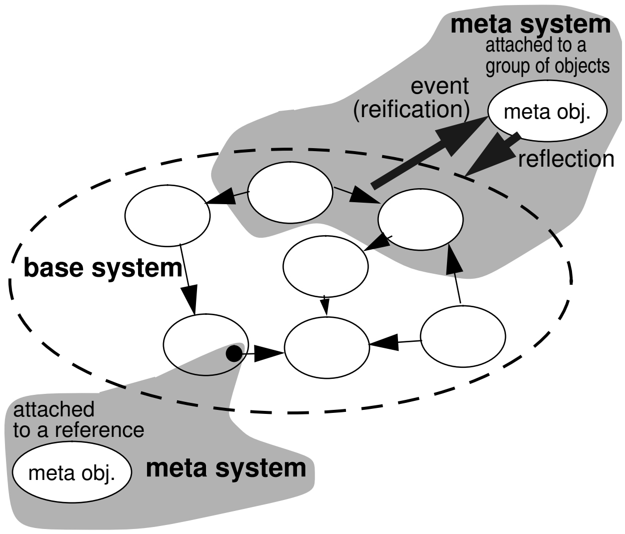


Figure 1: Computational model of behavioral reflection

3 Computational model

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

Our reflective approach is different. The system consists of the OS, the application program (the *base system*), and the *meta system*. The program may not be aware of the meta system. The computation in the base system raises events (see Figure 1). These events are delivered to the meta system. The meta system evaluates the events and reacts in a specific manner. All events are handled synchronously. Base-level computation is suspended while the meta object processes the event. This gives the meta level complete control over the activity in the base system. For instance, if the meta object receives a *method-enter* event, the default behavior would be to execute the method. But the meta object could also synchronize the method execution with another method of the base object. Other alternatives would be to queue the method for delayed execution and return to the caller immediately, or to execute the method on a different host. What actually happens depends entirely on the meta object used.

In our current implementation a computation only raises an event if one of its methods is called. We plan to include more event types in a later version of MetaJava. One could imagine events for outgoing method calls, variable accesses or object creations.

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

Not every object must have a meta object attached to it. Meta objects may be attached dynamically to base objects at run time. This is especially important if a distributed com-

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

MetaObject ***findMeta*** (Object base)
Find the responsible meta object for a base object.

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

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

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

void ***createStubs*** (Object obj, String methods[])
Create method stubs for the base object, which replace the specified methods. If such a method is called, the stub delegates control to the event-handler method of the attached meta object.

Object ***cloneReference*** (Object ref)
Create a new reference which points to the same object as the one given. This method is used to attach meta objects to references.

Class ***cloneClass*** (Object ref)
Clone the class of the object given. The clone becomes the class of this object. This method is used when a meta object is attached to an object to avoid interference with other instances of the same class.

Figure 2: Selected methods of the meta interface of the MetaJava virtual machine

putation is controlled at the meta level and arbitrary method arguments need to be made reflective. As long as no meta objects are attached to an application, our meta architecture does not cause any overhead. So applications only have to pay for the meta-system functionality where they really need it.

A meta object can be attached to a reference, an object or a class. If it is attached to an object, the semantics of the object is changed. Sometimes it is desirable only to change the semantics of one reference to the object — for example, when tracing accesses to the reference, or when attaching a certain security policy to the reference [24]. Attaching a meta object to a class makes all instances of the class reflective.

To fulfill its tasks the meta object has access to a set of methods which can manipulate the internal state of the virtual machine. These methods are called the *meta interface* of the virtual machine. Only the MetaObject class, its subclasses, and members of the package meta can invoke these methods. A list of the most important methods of the meta interface is given in Figure 2.

4 Implementation issues

Integration into Java. We implemented the meta interface methods as a collection of native methods that reside in a dynamic link library. Extending the virtual machine this way is common practice. Sun has used this technique with the network package java.net and the UNIX library libnet.so (Figure 3).

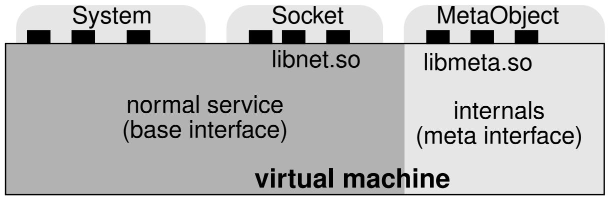


Figure 3: Extension of the Java virtual machine with a meta interface

Changes to the Java object structure (Figure 4)**.** To reify incoming method calls (i. e., to pass control to the meta system), the object's method table is replaced by a new one that contains stub procedures in the place of the original methods (Figure 5). The original methods are saved at the end of the method table. To avoid effects on non-reflective objects of the same class, the class block has to be duplicated. The superclass pointer of the reflective class block is modified to point to the original class block. This makes the reflective object type compatible with the original object.

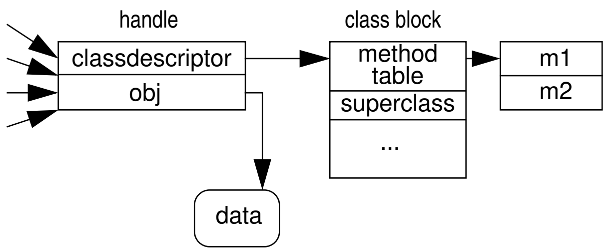


Figure 4: Structure of a Java object (simplified)

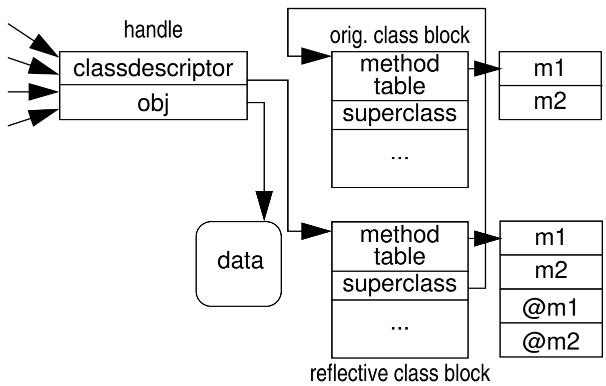


Figure 5: Structure of a reflective object

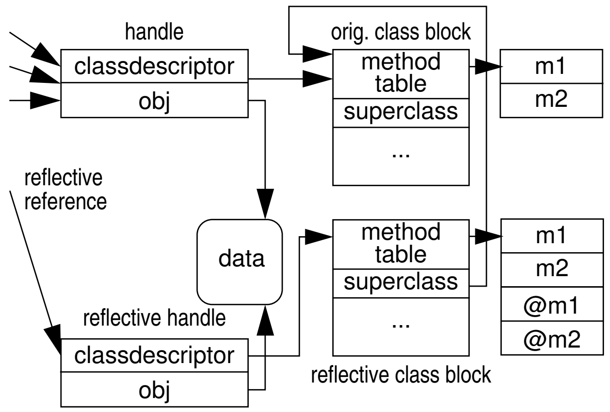


Figure 6: Object structure for a reflective reference

To attach a meta object to a reference, the object handle is duplicated using the cloneReference method of the meta-interface and the reference is redirected to the reflective handle (Figure 6).

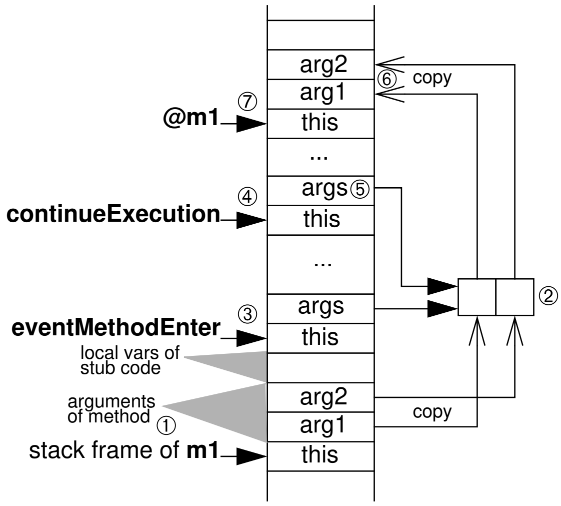
These modifications of the object structures cause some problems. The identity of objects is checked by the virtual machine by comparing the handle pointers. If a meta object is attached to a reference, as shown in Figure 6, the difference between the references becomes visible at the base level. This violates transparency but can only be changed by modifying the compare–byte-code evaluation in the interpreter. To check the identity of Java objects the interpreter should not compare handle pointers, but the pointers to the data area in the handle.

Stub processing. Figure 7 describes the passing of parameters between base level and meta level and the corresponding stack layout. It also shows the efficiency of the mechanism: An empty meta object, which passes control immediately back to the base level, adds the cost of two additional method invocations and the cost of allocating a buffer for the method arguments.

5 Use of meta objects

5.1 Tracing method invocations: MetaTrace

The class MetaTrace, listed in Figure 8, 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. If one wants to



When method m1 is invoked, the stack contains the this-pointer of the called object and the method arguments ①.

The stub code allocates a new array of object references ② and copies the arguments into it. It then calls the event-handler method eventMethodEnter ③ of the meta object and passes a reference to the argument buffer (args).

The computation continues on the meta level now. Finally the meta object may decide to execute the original base-level method with a call to the meta-interface method continueExecution ④. The args pointer is passed ⑤ to allow reconstruction of the argument list ⑥ before invoking the saved original method m1 (@m1) ⑦.

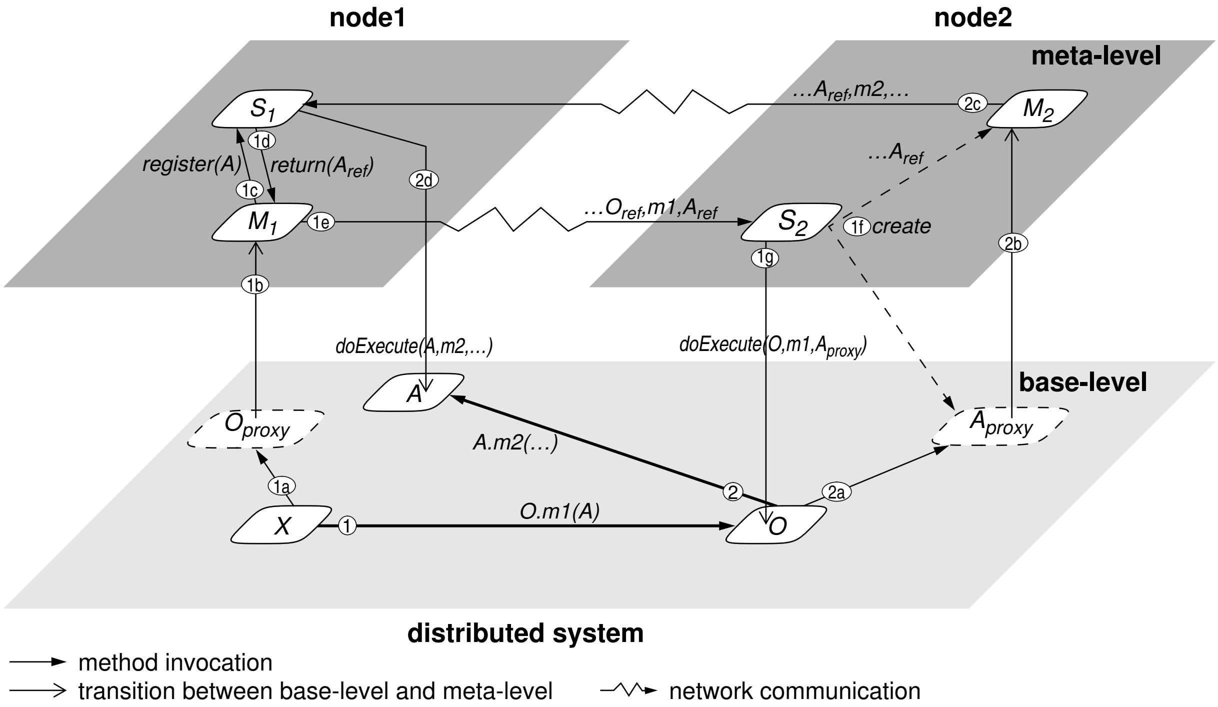
Figure 7: Invocation of a reflective method

```
public class MetaTrace extends MetaObject
{
    public void attachObject(Object o, String methodnames[] ) {
        createStubs(o, methodnames);
        attachObject(this, o);
    }

    public void eventMethodEnterVoid(   EventMethodCall event) {
        System.out.println("Method" + event.methodname + "called!");
        System.out.println("Signature" + event.signature);
        for(int i=0; i< event.n_args; i++) {
            System.out.println("Arg:" + event.arguments[i].toString());
        }
        continueExecutionVoid( event);
    }

    public Object eventMethodEnterObject(
                                        EventMethodCall event) {
        System.out.println("Method" + event.methodname + "called!");
        System.out.println("Signature" + event.signature);
        for(int i=0; i< event.n_args; i++) {
            System.out.println("Arg:" + event.arguments[i].toString());
        }
        Object ret = continueExecutionObject(event);
        System.out.println("returned:" + ret.toString());
        return ret;
    }
}
```

Figure 8: The implementation of the tracing meta object



(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 server S_1 and (1d) obtains the location-independent handle A_{ref}. (1e) It calls server S_2 with this handle, the method name, and the handle O_{ref} for the base object *O*. (1f) Server S_2 installs the proxy A_{proxy} for the argument object at node 2 and attaches meta object M_2 to it. (1g) Server 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 9: Invocation of remote methods with reference passing

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.

5.2 Remote method invocations: MetaRemote

Figure 9 shows what happens when an object invokes a method of a remote object. A program with similar behavior is given in Figure 10. 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 [20] or in CORBA environments [23], 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 meta object 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.

Performance tuning. Special meta-level implementations are outside the focus of this paper, 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 all primitive types (byte, char, enum, float, double, int, long, short, boolean) and String to the remote node instead of passing references. Copying mutable objects would require a coherence protocol between the replicas. 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 [29]. Hints for the optimal strat-

```
package meta.test.remote.test0;
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.attachNewInstance(
                                  "meta/test/remote/test0/Callee");
        Arg a = new Arg();
        s.test(a);
        System.out.println("called a.get(): " + a.get());
    }
}
```

Figure 10: Application of the MetaRemote meta object

egy for parameter passing (shallow or deep copy) could be provided by the base level with explicit meta interaction [17]. 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 meta object. Migration and copying would need an extended meta interface (Figure 2), which supports access to the object's state.

5.3 Further proposals for meta objects

Migration control. Different strategies to control the migration of objects, e.g. the computational field model [27], can be implemented at the meta level as shown in [22].

Object clustering. Grouping of objects can sometimes provide more advantages than keeping objects and their memory location orthogonal [6]. 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.

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 [26] use meta-object protocols to implement atomic data types with different concurrency-control algorithms.

Synchronization. Meta objects can be used to implement generic synchronization schemes — for example, constraining the sequence of events with path expressions, as proposed in [2]. Lopes and Liebeherr [16] 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.

Extended transactions. The classic ACID model of transactions is not sufficient for some application domains, especially for reactive applications [8]. To make extended transaction models practical to use, Barga and Pu [1] propose a Reflective Transaction Framework which transparently assigns extended semantics to transactions.

Fault tolerance. Fabre and colleagues show how different fault-tolerance mechanisms can be implemented at the meta level [10].

Just-in-time compilation. The meta interface can be extended by the addition of methods to allocate native code buffers and to execute native code in such a buffer. Such methods may be employed by a special meta object which, when receiving a method-enter event, dynamically compiles the byte code of the base-level method into the buffer and then executes the compiled method.

6 Related work

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 [12], [28]. OpenC++ Version 2 [3], [4] is a compile-time MOP 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.

AL-1/D. AL-1/D [21] is a Smalltalk-based language for distributed computing. AL-1/D partitions the meta level into different views of the base system to facilitate programming in a distributed environment. This 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 easily be implemented on the meta level.

Coda. Coda [19] uses fine-grained decomposition into different concepts of base objects (send, receive, accept, queue, protocol, execution, state). 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.

Moostrap. Moostrap [5] is a prototype-based language. Method execution is split into two phases: *lookup* and *apply*. Lookup is similar to our event mechanism, whereas apply resembles the doExecute meta-interface method.

7 Project status and future work

The meta objects MetaTrace and MetaRemote have been implemented as described. Furthermore a simple just-in-time–compiler meta object has been implemented. The meta interface has been extended with several methods to support a fully reflective architecture, which allows inspection and modification of all aspects of classes and objects (e.g. reading and writing the byte code of methods, modifying the constant pool of a class, reading the layout of the instance variables, etc.).

Our current implementation still has some limitations. These limitations are mainly due to the primitive data types of Java, which are not subclasses of Java's Object Class. Methods that have primitive types as arguments or return types can not be reified yet. This will be changed in a later version of MetaJava where primitive types will be packed into their corresponding wrapper type (e.g. Integer for int).

We are currently experimenting with the composition of meta objects and the configuration of the meta system. One major issue is to develop concepts to make the attachment of the meta objects to software modules configurable and to keep such configuration statements out of the functional code of the application program.

Another very important topic, which has not been addressed in this paper, is the aspect of security: It is obvious that, by providing the meta-level interface as we described it, the user can gain control over any component of the run-time system. Mechanism to avoid misusage have to be provided by a security architecture.

Further information about the project status can be obtained from http://www4.informatik.uni-erlangen.de/IMMD-IV/Projects/PM/Java/.

8 Conclusion

Adding a meta-level architecture to the Java virtual machine opens the Java environment for a broad range of run-time system extensions. We have described the mechanisms to achieve these extensions. As in all other open operating system architectures, the remaining issue is to develop concepts that allow comfortable configuration and control of these mechanisms without muddling the functional code of the application software.

9 References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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/).

[15] A. Lindström. 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.

[16] 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.

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Rev. 2.0, July 1995.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.