Reflections on Remote Reflection

Michael Richmond
Macquarie University
mar@ics.mq.edu.au

James Noble
Victoria University of Wellington
kjx@mcs.vuw.ac.nz

Abstract

The Java programming language provides both reflection and remote method invocation: reflection allows a program to inspect itself and its runtime environment, remote method invocation allows methods to be invoked transparently across a network. Unfortunately, the standard Java implementations of reflection and remote method invocation are incompatible: programmers cannot reflect on a remote application.

We describe how Java systems can be extended to support ***Remote Reflection*** *transparently by extending the standard Java API. Remote reflection can support remote debuggers, performance monitors, programming environments, application component servers such as Enterprise JavaBeans, and any other Java system that can be distributed across a network.*

1. Introduction

Programming and networking have been converging steadily since the explosion of the Internet in the middle of the 1990's. The Java programming language [5] is the most visible expression of this convergence. Java is designed so that programs can be transported across networks and executed in a Virtual Machine, unlike many other languages, regardless of the actual architecture and operating system of the machines on which it runs.

Java specifically supports distributed programming through the Remote Method Invocation (RMI) facility [18]. This allows a program on one machine to use objects and programs on remote machines. For example, in a system used to sell tickets to arts performances and sporting events a Java Applet running in a web browser could take users' orders, then use RMI to send ticket details to a central transaction server across the Internet.

As modern programming language, Java also includes support for Reflection [2, 4, 10]. Normal programs (or *base-level* programs) manipulate objects and values that refer to an external domain in the real world. In contrast, reflective programs (*meta-level* programs) manipulate objects and values from base-level programs [8, 10, 14]. For example, a base-level program to process ticket sales would have objects that represented tickets, events for which tickets can be issued, and patrons who purchase tickets. A meta-level program would have objects that represented the classes, methods, fields, and interfaces of the base-level program.

Reflection is often used to support programming tools such as debuggers and performance monitors. Such a tool, using reflection, can gather all the information it needs about the classes, class attributes, and objects making up the program being debugged or monitored. Reflection is also used in component-based systems, such as JavaBeans [16] or Enterprise JavaBeans [15], so that component containers can adapt themselves to the details of the components they will host.

In theory, reflection and remote method invocation should be orthogonal — a program should be able to use reflection and RMI separately or together without any negative interaction between the two facilities. Unfortunately, in Java, remote method invocation and reflection cannot be used together. The meta-level of a Java program can only reflect on local objects, that is, objects within the same Java virtual machine. The failure to support remote reflection has several consequences in practice:

- Debuggers cannot directly debug programs running in remote JVMs,
- Program visualisation systems cannot visualise remote programs,
- Component servers (such as Enterprise JavaBeans) cannot be managed or debugged remotely.

This paper addresses this problem by describing how Java can be extended to support remote reflection. The paper is organised as follows: the next section introduces the Java mechanisms for reflection and remote method invocation, providing a brief overview of the application programmer interface of each. Section 3 then describes the problems with Remote Reflection in Java and presents our design for a solution. Section 4 then discusses issues and recommends Java API changes to support transparent Remote Reflection.

```
Account anAccount = new Account(...);

Class   acClass = java.lang.Class.forName("Account");
Method  deposit = acClass.getMethod("deposit",[Class.forName("int")]);

deposit.invoke(anAccount, [250]);

Field   balField = acClass.getDeclaredField("balance");
```

Figure 1. Java Reflection code fragment.

Section 5 places this work in context of related work, and Section 6 concludes the paper.

2. Background

2.1. Java Reflection

The Java Reflection application programmer interface (API) allows a running program to retrieve information about itself and the runtime system (virtual machine) in which it executes [17]. Using reflection, a program may obtain information about its structure and the runtime environment. With this information the program can instantiate arbitrary classes, invoke methods, or alter data fields — bypassing any scoping rules or protection boundaries. This makes reflection a powerful but potentially dangerous language mechanism. The ability to circumvent call stack scoping and protection boundaries does, however, allow certain types of applications to be implemented orthogonally to the remainder of the program logic. This is particularly useful when developing applications like debuggers, monitors, and persistence mechanisms.

Reflection is supported in Java as part of the standard Java API. This API provides classes that represent array, class, method, field, constructor, and object instances in an executing Java Virtual Machine (JVM). Because these classes are about the program itself they are known as *meta-classes*, and their instances known as *meta-objects* [8].

A fragment of code, that uses Java reflection, is shown in Figure 1. In this example a new `Account` object is first created and a reference to the `Class` object corresponding to the `Account` class is obtained. This `Class` object is then used to locate the `Method` object for a `deposit()` method, which takes an integer as its argument, in the `Account` class.

A deposit is then performed on the `Account` object by invoking the `deposit()` method using an instance of the `Method` meta-class. Finally, the code obtains the reflection class representing the `private` field `balance` in the `Account` class.

Since reflection is inherently dangerous it is obvious to question the inclusion of it in a language: an unscrupulous programmer can use reflection to destroy the integrity of the base-level program. To avoid these problems, the Reflection API supports a property, `ReflectPermission`, in the JVM environment which disables reflection within that virtual machine. This is enforced by the `java.lang.reflect.AccessibleObject` super meta-class in concert with any existing security manager.

2.2. Remote Method Invocation

The Java Remote Method Invocation API is the basis for distribution support in Java. RMI provides location transparent object references, and automatic argument and return value marshalling to provide method invocation seamlessly across network and JVM boundaries.

When developing distributed applications, operations over the network should appear to behave identically to operations on local objects. In practice this is not completely possible, as operations over the network have more failure cases than local operations. These differences are unavoidable due to the nature of network communications. The goal of network transparency however, is to minimise these differences as much as possible.

RMI provides network transparent method invocation for objects that are declared as implementers of the `java.rmi.Remote` interface. This interface defines no method signatures, rather, it is used to mark the object as being remotable. It also provides type equivalence between the implementation of the remote object class, and the stub classes generated to act as local proxies for the remote object.

Objects are type equivalent if it is type-safe to use an object of one type in place of an object of another type. In the case of RMI both the implementation of the remote object and the generated stub object implement the same interface: i.e., the interface defined for the remote object. In the Java type system, this means the client of a remote object can define a variable to hold a reference to an object whose type is the remote interface of the remote object. As long as the object is accessed through this variable the client will only see an object of the remote interface type.

Remote operations via RMI are constrained to those that may be performed on a standard Java interface, that is, the

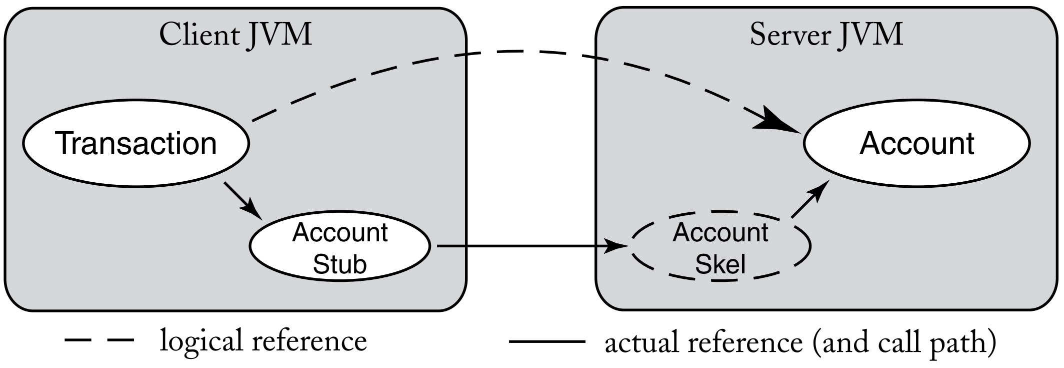

Figure 2. Overview of Java Remote Method Invocation model.

RMI API does not support client access to fields of remote objects, nor does it support access to static fields or methods in the remote class.

A functional overview of the RMI call path is shown in Figure 2. In this example, the `Account` object is providing a service which is accessed by the `Transaction` object. Each of these objects are located in different JVMs, potentially on separate hosts. In the case of both objects residing in a single JVM the RMI sub-system is able to invoke the method directly, thus bypassing the marshal/unmarshal process.

RMI allows the developer to act as though the client holds a reference directly to the server object. This *logical reference* can be treated like any other local reference. In our example, the `Transaction` object logically holds a reference directly to the `Account` object.

In reality, the reference held by the client refers to a *stub* object in the local JVM. The implementation of this stub is generated at compile time by `rmic` — the Java RMI stub compiler. This stub object performs the marshalling of invocation arguments, that is, it collects the Java objects and encodes them so that they can be transmitted across the network. The stub then communicates with the server, unmarshalling any return value. Importantly, the stub also implements the remote interface of the remote object, thereby ensuring the stub's type is equivalent to the remote object's type. (In this case, the `Account Stub` will effectively have the same type as the `Account` object since the stub implements the `Account` class's remote interface).

The stub object communicates with a matching skeleton object in the server's JVM. This skeleton is generated by `rmic` at the same time as the stub class. Starting with version 1.2 of the Java SDK, these skeleton classes are generated at run-time by the server JVM if they cannot be found by the server. The skeleton object is responsible for unmarshalling any invocation arguments, performing the local method invocation on the server object, and marshalling any return value.

Under RMI all invocation arguments are passed by copy to the server JVM. After the method call has completed only the explicitly declared return value is returned to the client. As a result, the server developer must be careful to ensure that they do not rely on side effects in the server object.

When passing structural objects such as arrays, vectors and trees it is important to remember that only the root object is passed by RMI. The developer must ensure that the elements contained within these structures are remotable objects, otherwise, fatal exceptions will be raised on the client during unmarshalling.

Figures 3 and 4 show an example of the code required to make a Java object accessible using RMI. The programmer must create an interface for the object to act as the remote interface for the object. This interface must extend the `java.rmi.Remote` interface and declare all methods to throw `java.rmi.RemoteException`. If an error occurs during the remote method invocation an exception will be returned to the client that is a sub-class of `java.rmi.RemoteException`.

The implementation of the remote object must be declared to as implementing the previously defined remote interface. Each public method in the remote interface must be implemented in this class and declared as throwing `java.rmi.RemoteException`.

3. Remote Reflection

The ability for a program to examine itself via reflection can be extremely useful when developing certain types of applications. When using a monitor or debugger to observe a program it is desirable to minimise the impact that the observer has on its subject. The introduction of any additional process into a system will potentially alter the behaviour of the program being observed. With multi-threaded and synchronous programs this is particularly evident, with additional processes altering the scheduling behaviour of the system.

```
import java.rmi.*;
import java.net.URL;

public interface Account extends java.rmi.Remote {

   public void deposit(int amount) throws RemoteException;

   public int getBalance() throws RemoteException;
}
```

Figure 3. Java RMI interface code fragment for `Account.class`.

```
import java.rmi.*;
import java.rmi.server.RMIClassloader;

public class AccountImpl implements Account {

   private int balance;
   ...
   public void deposit(int amount) throws RemoteException {
      balance += amount;
   }
   ...
}
```

Figure 4. Java RMI object implementation code fragment for `AccountImpl.class`.

For distributed systems, it is preferable use a single debugger or monitor to observe the entire system from one place rather than being tied to the location of each component in the system. To achieve this the debugger must know how to observe over a network, in addition to observing local components. In RMI we have a mechanism which allows remote operations to be performed as if they were local operations. Rather than re-implement a remote operation mechanism, it makes sense to use RMI to access the remote components in the system.

When implementing such a debugger, the programmer must avoid reliance on language features not supported by the RMI model. Specifically this means that all objects must define interfaces, no static methods are used, and object fields are accessed via get and set methods rather than directly.

As long as these constraints are observed, the conversion of a local debugger or monitor into one which can act remotely should be trivial. Unfortunately, the Java reflection API restricts reflection to the local JVM.

3.1. The Problem

The current implementation of the Java Reflection API (SDK 1.3) restricts reflection to the local JVM state. In spite of the RMI model making the question of locality orthogonal to the class implementation in most cases. This inability, to use reflection to act upon remote objects, breaks the RMI goal of providing network transparency to developers of distributed systems, thus, preventing developers from using reflection to develop remote and distributed monitors, debuggers, and visualisation tools.

There are a two main reasons for this restriction. The first is that the reflection API relies on the static methods `Class.forName()` to provide instances of reflection objects. If we are to provide remote reflection, then some way of remotely calling these static methods must be provided.

The second, and most important reason, is that the core reflection classes are defined as `final` in the standard Java API, thus preventing the programmer from sub-classing to extend functionality.

3.2. Final Classes

With most Java classes it is a trivial to use sub-classing to make them remotable. The Java keyword `final`, however, prevents programmers from sub-classing a particular class. There are three possible reasons for this decision.

First, there is the need to prevent programmers from sub-classing the reflection classes to expose their constructor methods. In a fully reflexive language, it is possible to alter the language semantics by over-riding parts of the reflection classes, for instance, over-riding `Method.invoke()` to change the semantics of method invocation. In Java, the reflection classes are *representations* of the underlying virtual machine state rather than the actual JVM state. It is therefore not possible to use reflection in Java to alter the language behaviour. Furthermore, it is necessary to ensure that only the JVM is capable of creating reflection objects. Oth-

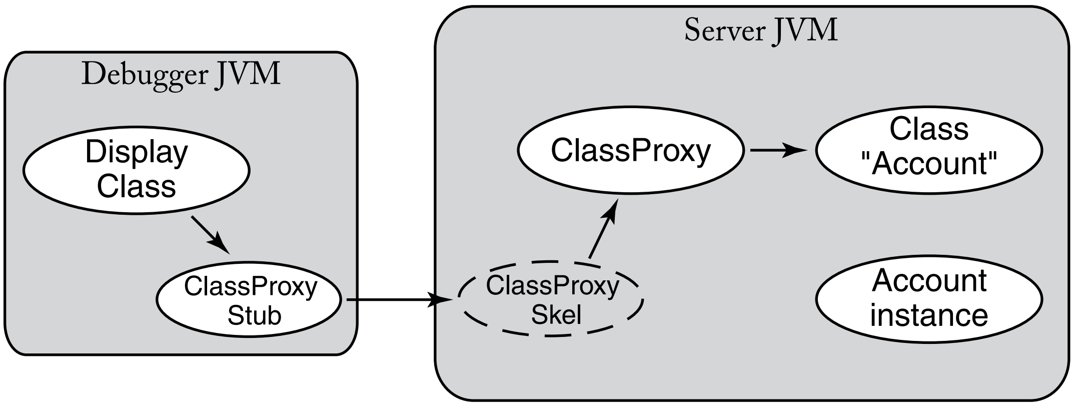

Figure 5. Proxy model of remote reflection.

```
...

Class MethodProxyImpl implements ClassProxy {
   Method method;

   public ClassProxy getDeclaringClass() throws RemoteException {
      return (ClassProxy) new ClassProxyImpl(method.getDeclaringClass());
   }
...
```

Figure 6. Remote Reflection proxy code fragment.

erwise, the integrity of the Java Reflection mechanism would be undermined.

The second reason is that reflection classes typically access the JVM data structures directly through native methods. The result is a tight coupling between meta-objects and base-objects, binding the reflection implementation to a specific JVM. This constrains meta-objects to exist in the same JVM as the base-objects they reflect upon.

Finally, there is the view that reflection allows you to inspect, reason and act at the meta-level of a program. With the presumption that access to this meta-level will only be required from within the program being reflected upon.

3.3. Solution

To bypass these problems, we have written a set of proxy [3] classes which mirror the method signatures of the Java reflection classes. These proxies reside in the same JVM as the standard reflection meta-objects, i.e., in the same JVM as the base-level objects being reflected upon. Unlike the standard meta-objects, these proxies are remotable (they implement the `java.rmi.Remote` interface) and so can be referenced remotely. (See figure 5.)

Our proxy classes do not fully solve the problem of providing Remote Reflection in Java. They do, however, provide an effective practical solution, while helping us identify necessary changes to the Java API. These changes are presented in the section 4.1 of this paper.

3.4. Implementation

The implementation of these proxy classes is quite simple. One proxy class exists for each Java reflection meta-class. At run time, each proxy object holds a reference to the meta-object that it proxies. The proxy class implements every method in its corresponding meta-class, by calling the appropriate method of the meta-object it represents (as shown in Figure 6). For the methods where the reflection class returns an array of reflection objects, the proxy is responsible for generating a matching array of proxy objects to be returned to the reflection client.

The reflection API is fairly self contained, with reflection objects being used mainly by the reflection API. This means our proxy classes need not be type equivalent with the classes they represent, rather, they form a parallel proxy API. In any event, type equivalence is prevented by the declaration of the reflection classes as `final`. The drawback is that remote reflection references cannot be used within the reflected JVM after being passed there via some channel.

It's important to note that this implementation is solely within the Java language, and does not require any modifications to the Java Virtual Machine. This has several advantages. Most importantly, the remote reflection mechanism will operate on any JVM that implements the standard RMI and reflection interfaces. The remote reflection mechanism can be loaded into any running JVM, without requiring the JVM to be restarted, reloaded, or re-compiled. Java language-level programming is also much simpler than modifying a JVM.

The drawback is that our implementation is not completely transparent. Our proxy objects can be used only in calls to other proxy objects. That is, we cannot use these proxy classes in calls to the standard Java reflection API. Additionally, some well known home for the `Class.forName()` methods is required in the remote JVM to act as an object factory [3] for the `ClassProxy` instances. This is a result of RMI not supporting static methods in remote classes.

In our project we were sure that proxy references and reflection API references would not be combined in a method invocation. In general use, however, the programmer must ensure that any proxy references transferred to the remote JVM are not used in calls to the reflection API. This will be caught by the type checker during compilation, but ideally a remote reference to a reflection object could be used in any call to the reflection API. The changes to the Java Reflection API we suggest in section 4.1 remove this distinction between remote and local reflection references.

Additionally, programmers must not expect object equality based on identity between reflection references and reflection proxy references. Rather, the equality of these references should be based on object value. For example, it is reasonable to test if the local `Class` object for `java.lang.String` is equal to the remote `Class` object for `java.lang.String`. This will fail, however, when using our proxies to access the remote `Class` object, since the programmer is comparing a `Class` object to a `ClassProxy` object.

4. Discussion

The use of proxies to make Java reflection accessible from a remote machine has been successful. They have allowed us to use an existing object visualisation tool to inspect, and act on, a remote Enterprise JavaBeans server. For general use, however, some limitations remain. Namely:

- the reflection proxies are not type equivalent to the reflection classes they represent, so cannot be interchanged in method calls,
- some object is required, on the remote host, to act as an object factory [3] for `ClassProxy` instances,
- extra overhead is introduced by the proxy objects,
- object equivalence between reflection proxies and reflection classes is not provided.

All but the last of these limitations can be solved by altering the Java Reflection API to support RMI. In effect collapsing the proxies into the reflection objects themselves. This changes the last limitation into one of class equivalence. Class equivalence is considered in section 4.3 of this paper.

4.1. Changes to the Java API

The first problem with the Reflection API is that none of the classes it defines are remotable. One solution is to not define the API classes as `final`. Thus allowing programmers to use the RMI metaphor of sub-classing to implement the empty `java.rmi.Remote` interface to make the class remotable. Unfortunately, this would allow programmers to sub-class to expose the constructors of the reflection API.

4.2. Reflection API Changes

A better solution is to change the reflection API so the constructors are declared as `final`, allowing API extensions without exposing the object constructors. The Java language however, does not allow constructors to be declared `final` [5].

Our preferred solution is to make the API classes implement the `java.rmi.Remote` interface. Every Java reflection class, with the exception of `java.lang.Class`, already inherits from `java.lang.reflect.AccessibleObject`. As `Class` and `AccessibleObject` do not explicitly inherit from any other class, this would not disrupt the existing class hierarchy. Additionally, `AccessibleObject` will need to implement the empty `java.io.Serializable` interface to comply with RMI API requirements.

These changes will allow remote access to the Java Reflection objects, ensuring type equivalence between remote and local references. They will also prevent programmers from implicitly reflecting on the RMI stub objects, effectively by-passing them during RMI method invocation. If it is necessary to reflect on the stub and skeleton objects the programmer can still use `instanceOf()` and appropriate type casting to explicitly reflect upon these classes.

A second issue with these API changes is that RMI does not support static methods on remote objects. In the Reflection API, every reflection class is effectively instantiated via a `Class` object. These `Class` objects are instantiated by calling either `java.lang.object.getClass()` or the static `java.lang.Class.forName()` methods.

To support remote instantiation of `Class` objects it is therefore necessary to modify the `Class` class and the JVM to allow the programmer to specify whether a call to `java.lang.Class.forName()` is fulfilled in the local JVM or in a specified remote JVM.

4.3. Class Equivalence Problem

With these changes to the Java Reflection API, it will be possible to use reflection to inspect and act upon a remote JVM, achieving our original goal of allowing programmers to use Java Reflection remotely.

As previously noted, however, the problem of class equivalence remains. These API changes will make it possible for a programmer to instantiate a `Class` meta-object for the class `java.lang.String` in a remote JVM, and a second `Class` meta-object for `java.lang.String` in the local JVM. This is a problem: we will have two potentially unequal instances of the `Class` object in the distributed system. We refer to this as the *class equivalence problem*.

As discussed by Liang and Bracha it is possible for two instances of a user-defined class to exist within a single JVM. This may occur when a class is loaded by two separate class loaders. In this case the classes are not considered to be of the same type in Java since *a class type is uniquely identified by the combination of class name and class loader* [9].

In the local case, the loading of system classes is delegated to the Java system class loader which ensures only one meta-object for each class exists in the JVM. This is achieved by having class loaders delegate the loading of Java system classes to the system class loader [9], guaranteeing that all system meta-objects are unique. Within a single JVM all references to the `java.lang.String` meta-object will be equal on the basis of object identity. Without this guarantee the type safety of the system would be violated [9].

With remote reflection, however, each JVM has its own version of each class so there can be multiple instances of meta-objects, even for standard systems classes. In practice, however, only one of these meta-objects is the "real" meta-object for that JVM: because the remote meta-objects can only reflect upon base-objects from their own JVM, they cannot affect the integrity of any other systems.

5. Related Work

Neither reflection nor distribution are novel features of programming languages, whether or not they are object-oriented. Lisp, Smalltalk, Self, and Java are just a few of the reflective languages in common use today [2, 4, 5, 19]. Some of these reflective languages are very powerful indeed, for example, the CLOS object system is effectively implemented by a set of meta-objects that can be altered or replaced to change CLOS's behaviour [7, 8]. Emerald, Modula-3, Smalltalk, and Java once again are just a few of the languages with in-built in support for distribution, and thus remote invocation [1, 6, 13, 15, 18].

Furthermore, many languages combine reflection and distribution, often, as in CodA or ABCL/R relying on reflection to implement distribution [11, 12]. In particular, distributed versions of Smalltalk, one of the oldest object-oriented languages, have to solve many of the problems we encountered to support remote reflection [2, 13]. Due to the orthogonal nature of Smalltalk's design, this can be done much more easily that we have in Java. Indeed, our work can be understood as applying techniques developed for Smalltalk to Java, and understanding where they break down, as in the case of `final` and static methods, which are part of Java but are not part of Smalltalk.

6. Conclusion

Java Reflection and Remote Method Invocation should be orthogonal. A program should be able to reflect upon any object, local or remote. Any object, base-level or meta-level, should be able to be made remotable. In standard Java this is not the case: meta-objects from the reflection API cannot be accessed remotely. We have described how Remote Reflection can be implemented within the standard Java environment, without modifying the underlying virtual machine.

We have also recommended changes to the standard Java API which will allow Java Reflection to be used remotely while avoiding the limitations encountered when using a purely Java solution.

7. Acknowledgements

The authors wish to thank to thank Randy Smith for his assistance during the implementation of this work. We are grateful to SunLabs, California for their making this research possible through their student intern program.

Java, JavaBeans, and Enterprise JavaBeans are trademarks of Sun Microsystems.

8. References

[1] Birrel, A., Nelson, G., Owicki, S., & Wobber, E. Network Objects. *SOSP Proceedings*, 1993.

[2] Foote, B. & Johnson, R. E. "Reflective Facilities in Smalltalk-80", *OOPSLA'89 Proceedings*, ACM Press, October 1989.

[3] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.

[4] Goldberg, A. & Robson, D. "*Smalltalk-80: The Language and its Implementation*", Addion-Wesley, 1983.

[5] Gosling, J., Joy, B. & Steele, G. "*The Java Language Specification*", Addison-Wesley, 1996.

[6] Jul, E., Levy, H., Hutchinson, N., & Black, A. Fine-grained mobility in the Emerald system. ACM *TOCS*, V6 N1, 1988.

[7] Keene S. E. "*Object-Oriented Programming in Common Lisp: A Programmer's Introduction to CLOS*", Addison-Wesley 1989.

[8] Kiczales, G., des Rivieres, J. & Bobrow, D. G. "*The Art of the MetaObject Protocol*", MIT Press, 1991.

[9] Liang, S. & Bracha, G. "Dynamic Class Loading in the Java Virtual Machine", *OOPSLA '98 Proceedings*, ACM Press, October 1998.

[10] Maes, P. "Concepts and Experiments in Computational Reflection", *OOPSLA '87 Proceedings*, ACM Press, October 1987.

[11] Matsuoka, S., Watanabe, T., Yonezawa, A. Object-oriented concurrent reflective languages can be implemented efficiently. *OOPSLA' 92 Proceedings*, 1992.

[12] McAffer, J. Meta-level programming with CodA. In *ECOOP Proceedings*, 1995.

[13] McCullough, P. L. "Transparent Forwarding: First Steps", *OOPSLA '87 Proceedings*, ACM Press, October 1987.

[14] Smith, B. C. "Reflection and Semantics in Lisp", *Proceedings of the Principles of Programming Languages Conference*, ACM Press, 1984.

[15] Sun Microsystems, "*Enterprise JavaBeans Specification version 1.1*", Sun Microsystems, Palo Alto California, December 1999.

[16] Sun Microsystems, "*JavaBeans Specification version 1.01*", Sun Microsystems, Palo Alto California, December 1998.

[17] Sun Microsystems, "*Java Core Reflection Specification version 1.3*", Sun Microsystems, Palo Alto California, December 1999.

[18] Sun Microsystems, "J*ava Remote Method Invocation Specification version 1.3*", Sun Microsystems, Palo Alto California, December 1999.

[19] Ungar, D. & Smith, R. B. "Self: The Power of Simplicity", *OOPSLA '87 Proceedings*, ACM Press, October 1987.