Dalang - A Reflective Extension for Java

Ian Welch and Robert Stroud

University of Newcastle-upon-Tyne, United Kingdom NE1 7RU
{I.S.Welch, R.J.Stroud}@ncl.ac.uk,
WWW home page:http://www.cs.ncl.ac.uk/people/
{I.S.Welch, R.J.Stroud}

Abstract. Current implementations of reflective Java extensions typically either require access to source code, or require a modified Java platform. This makes them unsuitable for applying reflection to Commercial-off-the-Shelf (COTS) systems. In order to address this we developed a prototype Java extension Dalang based on class wrapping that worked with compiled code, and was implemented using a standard Java platform. In this paper we evaluate the class wrapper approach, and discuss issues that relate to the transparent application of reflection to COTS systems. This has informed our design of a new version of Dalang called Kava that implements a metaobject protocol through the application of standard byte code transformations. Kava leverages the capabilities of byte code transformation toolkits whilst presenting a high-level abstraction for specifying behavioural changes to Java components.

1 Introduction

We are interested in the problems of applying non-functional requirements to Commercial Off-the-Shelf (COTS) software components. In an environment such as Java components are usually supplied in a compiled form without source code, and can be integrated into a system at runtime.

Metaobject protocols [20] offer a principled way of extending the behaviour of these components. Metaobjects can encapsulate the behavioural adaptations necessary to satisfy desirable non-functional requirements (also referred to as NFRs) such as fault tolerance or application level security [1][30][31] transparently at the metalevel. Ideally we want to apply these metaobjects to compiled code that executes on a standard Java platform.

We reviewed available implementations of reflective Java that could be used to implement these metaobject protocols and found that none of them met our requirements. They either relied upon customised Java platforms or required access to source code. Accordingly we developed a prototype reflective Java extension called Dalang based on the standard technique of using class wrappers to intercept method invocation. The advantage of Dalang was that it did not require access to source code and was implemented using a standard Java platform.

Our subsequent experiences with Dalang\footnote{Dalang is the puppetmaster in Javanese wayang kulit or shadow-puppet plays.} have highlighted a number of problems with implementing reflection that arise from the approach of using class
wrappers to implement reflection, and more generally with attempting to apply reflection transparently to existing COTS software built from components.

We have applied these lessons to the design of the successor to Dalang called Kava\(^2\) which is based on wrapping not at the class level, but at the byte code level. Kava implements a runtime behavioural metaclass protocol through the application of standard byte code transformations at load time. Metalevel inter-captions are implemented using standard byte code transformations, and behavioural adaptation is implemented using Java metaclass classes. Although neither byte code transformation or metaclass protocols are new, what is novel is the implementation of metaclass protocols using byte code transformation in order to provide a higher level view of component adaptation than current byte code transformation toolkits currently provide.

The rest of the paper is organised as follows. In section two we provide a review of different approaches to implementing reflection in Java. Section three gives an overview of the class wrapper approach to implementing reflection. Section four presents an evaluation of the class wrapper approach. Section five discusses some of the problems of applying reflection transparently to existing applications. In section six we discuss the advantages of using byte code transformation to unify class wrappers and wrapped objects. In section seven we outline our design for a new version of Dalang called Kava that addresses a number of the problems raised in the two previous sections. In section eight we discuss the application of Kava. Finally in section nine we present our conclusions.

A prototype implementation of Dalang has been completed and is available from http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/dalang. We are currently developing an implementation of Kava, for further information see the project page at http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/kava.

2 Review of Reflective Java Implementations

In this section we briefly review a number of reflective Java implementations and attempt to categorise them according to the point in the Java class lifecycle where metalevel interceptors (MLIs)\(^3\) are added. Metalevel interceptors cause the switch during execution from the baselevel to the metalevel thereby bringing the base level object under control of the associated metaclass. Table 1 summarises the features of the different reflective Java implementations.

The Java class lifecycle is as follows. A Java class starts as source code that is compiled into byte code, it is then loaded by a class loader into the Java Virtual Machine (JVM) for execution, where the byte code is further compiled by a Just-in-time compiler into platform specific machine code for efficient execution.

All these implementations have drawbacks that make them unsuitable for use with compiled components or in a standard Java environment. Either they require access to source code, or they are non-standard because they make use of a modified Java platform. In order to address these drawbacks we implemented

\(^2\) Kava is a traditional South Pacific beverage with calming properties.
<table>
<thead>
<tr>
<th>Point in Lifecycle</th>
<th>Reflective Java</th>
<th>Description</th>
<th>Capabilities</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Code</td>
<td>Bean Extender [17]</td>
<td>Byte code preprocessor.</td>
<td>No need to have access to source code.</td>
<td>Restricted to Java Beans, requires offline preprocessing.</td>
</tr>
<tr>
<td>Runtime</td>
<td>MetaXa [14]</td>
<td>Reflective JVM.</td>
<td>Can intercept wide range of operations, can be dynamically applied.</td>
<td>Custom JVM.</td>
</tr>
<tr>
<td></td>
<td>Guaran [28]</td>
<td>Reflective kernel supported by modified JVM.</td>
<td>Interception of message sends, state access and supports metaobject composition.</td>
<td>Custom JVM.</td>
</tr>
</tbody>
</table>
a reflective extension called Dalang that required only access to compiled classes and use the standard Java platform. It was based upon standard class wrapper approach which we discuss in the next section.

3 Overview of Implementing Reflection Using Wrappers

In this section we give an overview of how class wrappers can be used to implement reflection in statically typed languages such as Java and give an overview of the implementation of Dalang.

3.1 Overview

The use of class wrappers or proxy classes in order to implement metalevel interception for method invocation is a common approach for adding reflection into a statically typed language such as C++[5] or Java [15][35]. This approach is similar to the Wrapper or Decorator pattern [13]. Figure 1 shows the collaboration diagram for the general case where a base level object has its method invocations intercepted and handled at a metalevel. Each role in the diagram is detailed below:

BaseObject. The class in this role has the behaviour of method invocations sent to it by a client adapted by the associated Wrapper MetaObject and any Concrete MetaObjects.

MetaObject. This abstract class takes the responsibility of defining the binding between a specific Wrapper MetaObject and the BaseObject, and implementing a general method for invoking methods of the BaseObject.

Wrapper MetaObject. This class extends the behaviour of each public method of the BaseObject class by redefining each method to invoke the respective method in the BaseObject class through a call to the invokeMethod method. Since each invocation is now handled by the invokeMethod method redefining this method will redefine the behaviour of each method invocation sent to the BaseObject.

Concrete MetaObject. There may be any number of Concrete MetaObject classes that are invoked by the Wrapper MetaObject in order to extend the behaviour of the BaseObject class.

The main differences between class wrapper implementations of reflection are the way that class wrappers are generated and how they are substituted for the base object class.

Dalang exploits the reflective hook [22] into the Java class loading mechanism provided by user-defined class loaders. User-defined class loaders can be created by subclassing java.lang.ClassLoader and implementing a custom loadClass() method. Classes are then loaded by the user-defined class loader either explicitly by application code invoking loadClass(), or by the JVM implicitly invoking loadClass() to load a class referenced by a class previously loaded by the user-defined class loader. Dalang uses a special reflective class loader (dalang.ReflectiveClassLoader) that transforms classes as they
Fig. 1. Class collaboration diagram for class wrapper approach to implementing reflection.

are loaded into the JVM. At load time Dalang uses the Java Core Reflection API (JCR) in order to discover the public interface of the Base Object. It then generates source code for the Wrapper MetaObject which is dynamically compiled. The generated Wrapper MetaObject is then switched for the BaseObject through class renaming at the byte code level.

3.2 Dalang Implementation

In this section we give an overview of how Dalang works, how it can be applied both statically to offline class files and dynamically at load time, and how the binding between metaobject classes and baselevel classes is specified.

Basic Approach Dalang implements metaobjects as wrappers for baselevel classes to allow interception of method invocations sent to the baselevel classes. This is a straightforward implementation technique used by many reflective systems but what makes Dalang novel is the use of the JCR rather than a preprocessor to derive the necessary type information. This means that Dalang can be applied post-compilation to the Java byte codes and does not require access to source code. Classes are wrapped in two steps. In the first step the baselevel class is renamed through modification of the class at the byte code level. In the second step a new class is generated that supports the interface of the baselevel class and the functionality of the metaobject class. A straightforward implementation would be to make the new class inherit from the base class and from the metaobject class. However, we cannot do this in Java because there is only single inheritance and the baselevel class could be marked as final which would prevent any other class inheriting from it. So the new class inherits from the metaobject
class but implements the interface of the baselevel class and its superclasses. The JCR is used to determine the interface of the baselevel class from the class byte code and perform a transitive closure over the interfaces of its superclasses. A new class with the same name and interface as the baselevel class is generated as Java source code and compiled into byte code using the standard Java compiler. This new class inherits from a specified metaobject class that defines how method invocations are handled. The actual method invocations are forwarded to the baselevel class. As the new class has exactly the same interface and the same name as the baselevel class it may be safely used by applications with no change required to their code. This new class essentially "wraps" the baselevel class. From this point on we will refer to this class as the wrapper class. For example, if the baselevel class was:

```java
public class C {
    public void f() {
        // do something useful
    }
}
```

Then the source code for the wrapper class will be as shown below:

```java
public class C extends MetaObject {
    private _C obj;
    public C() {
        obj = new _C();
    }
    public void f() {
        invokeMethod(obj, "f", null);
    }
}
```

This new class extends the metaobject class MetaObject. It has a pointer to an instance of the baselevel class which has been renamed _C, and provides implementations of the methods in the interface of the wrapped class together with any methods inherited from the parent metaobject class bound to the baselevel class. The constructor for the new class creates a new instance of the baselevel class and stores the pointer. The class MetaObject supports the method invokeMethod which reifies method invocation and allows its behaviour to be redefined. We exploit the method dynamic invocation facility of the JCR in order to perform the invocation. The MetaObject code that implements the dynamic invocation is shown below:
public Object invokeMethod(Object target,
    String method,
    Object[] arguments)
{
    Class targetClass = target.getClass();
    Object[] formalTypes = findTypes(arguments);
    Method m = targetClass.getMethod(method, formalTypes);
    return (m.invoke(target, arguments));
}

The first three lines deal with finding the class of the target object, finding the formal parameter types of the arguments, and using this information in conjunction with the name of the method to get an instance of the actual method to be invoked. The fourth line invokes the method on the target object with the specified arguments. Use of the JCR for dynamic invocation greatly eases the implementation of the `invokeMethod()` operation; otherwise Dalang would have to construct a large switch statement that contained an invocation of each method of the baselevel class. With this approach the operation for dynamic invocation can be logically separated from the wrapper and implemented in the `MetaObject` class. As shown the default behaviour of `invokeMethod()` is simply to invoke the named method on the wrapped class. In order to implement a different behaviour a new metaobject class should be defined that inherits from `MetaObject` and redefines `invokeMethod()`. For example, the following metaobject implements tracing behaviour on method invocations:

public class MetaTrace extends MetaObject
{
    public Object invokeMethod(Object target,
        String method,
        Object[] arguments)
    {
        System.out.println("before : " + method);
        super.invokeMethod(target, method, arguments);
        System.out.println("after " + method);
    }
}

With this wrapper whenever a method is invoked the name of the method is displayed before and after its invocation. Note that the `MetaObject` class still does the actual invocation of the wrapper method.

**Static and Dynamic Application** Dalang can be applied either statically to offline class files or dynamically as classes are loaded into the JVM.

*Static Application* When applied statically the wrapping steps described earlier are applied directly to class files. Dalang is invoked with the name of the class
to be made reflective and the metaobject class that is to be bound to the class. The class to be made reflective is first renamed both at the byte code level and file level, then the source code for a wrapper class with the original name of the baselevel class is generated. The wrapper class is then compiled using the standard Java compiler to generate a wrapper class file. Now, any class requesting the baselevel class will load the wrapper class instead.

**Dynamic Application** In order to apply Dalang at load time then class loading must be intercepted at runtime. We take advantage of the reflective hook [22] into the Java class loading mechanism provided by the class java.lang.ClassLoader. User-defined class loaders can be created by subclassing java.lang.ClassLoader and implementing a custom `loadClass()` method. Classes are then loaded by the user-defined class loader either explicitly by application code invoking `loadClass()`, or by the JVM implicitly invoking `loadClass()` to load a class referenced by a class previously loaded by the user-defined class loader. We define a special reflective class loader (Dalang.ReflexiveClassLoader) that transforms classes as they are loaded into the JVM. When a class is loaded using our reflective class loader then the processing is carried out as for the static case except that the renaming step occurs in-memory and the wrapper class is dynamically compiled. Our reflective class loader still guarantees the type safety and security properties of classes it loads and the wrapper classes that it generates on the fly. This is because although the user-defined class loader can redefine the fetching of the byte codes of a class, it cannot itself convert the byte codes into a runtime class object in the JVM. To do this it calls JDK native methods that request the JVM to perform the conversion. As part of this conversion process the byte codes are subjected to the Java verification process that guarantees type safety and security properties. Relatively few changes to an application are required to apply reflection at load time, especially if the application already makes use of a user-defined class loader. An example of the code required to explicitly load a class via a user-defined class loader is shown below:

```java
ClassLoader loader = new AppClassLoader();
Object myClient = loader.loadClass("Client").newInstance();
(Runnable)myClient.run();
```

An instance of the user-defined class loader AppClassLoader is created and then used to load the class Client. An instance of the Client is then created by invoking `newInstance()` on the newly loaded class. It is assumed that Client implements a locally defined interface Runnable that allows casting to take place and the run method of Client to be invoked. Any class referenced by Client will be loaded by the JVM using the class loader used to define Client that in this case is an instance of MyClassLoader. This is exactly the approach used by browsers to load applets. The system shown in fig. 2 uses a reflective class loader in conjunction with an existing application class loader to perform some form of custom loading of classes. The change made to the application in order to use Dalang was simply that the application now requests the reflective class loader
**ReflectiveClassLoader** to load classes instead of the baselevel application class loader. The changed code is shown below:

```java
ClassLoader loader = new ReflectiveClassLoader(new AppClassLoader());
Object myClient = loader.loadClass("Client").newInstance();
(Runnable)myClient.run();
```

In the code example shown above **ReflectiveClassLoader** wraps the existing user-defined class loader. It does not use the wrapped class loader to load the class but to fetch the class byte code. The **ReflectiveClassLoader** renames the class, requests the JVM to install the byte code as a runtime class object, uses the JCR to determine the interface of the class, generates the wrapper class as source code, dynamically compiles it and then requests the JVM to install the resulting byte code instructions as a runtime class object. If no user-defined class loader is already being used then the application must be changed to perform explicit loading of classes by the **ReflectiveClassLoader**. In this case the **ReflectiveClassLoader** will follow the standard Java class loader behaviour of loading classes from the directories specified in the CLASSPATH.

![Diagram](image)

**Fig. 2.** Overview of Architecture. Shaded parts indicate Dalang. Non-shaded parts are the standard Java runtime system.
3.3 Binding the Baselevel Object to MetaObject

As we do not have access to the source code of the classes being made reflective we cannot take the approach of some languages and use source code annotations to specify bindings between objects and metaobject classes. Instead Dalang uses a metaconfiguration file (Meta Config) to determine which metaobjects apply to which classes and methods. The metaconfiguration file must exist in the same execution environment as the application.

4 Evaluation of Class Wrapper Approach

In this section we evaluate and discuss implementation of reflection using the class wrapper approach. We focus on the following areas:

1. Wrappers
2. Inheritance
3. Security Considerations of applying Metaobjects
4. Class loaders

4.1 Wrappers

In a paper on using class wrappers for adapting independently developed components [16], Holzle makes the point that problems occur when the wrapper object and the object being wrapped are implemented separately as in this approach. He explains that such an approach to wrapping suffers from two major problems: the "self problem", and the encapsulation problem. The identification of these problems is not new but they have not previously been discussed in relation to the implementation of reflective systems. The "self problem" was first described by Lieberman [23], Lieberman asserts that you cannot use inheritance based languages such as Java to implement delegation. The problem is with the rebinding of the self\(^3\) variable that takes place when the method invocation is delegated. It is possible to work around this by passing the self variable as an additional argument in all method invocations and making all self invocations use this passed variable instead of the default. This requires that the classes follow a particular programming convention. However, since the classes being wrapped were constructed independently of their use with class wrappers it is unlikely that they would support such a convention unless we could transform the compiled classes’ methods.

Figure 3 illustrates the problem. In the left hand case where true delegation takes place all method invocations are intercepted by the wrapper for A whilst in the right hand case only the first method invocation is intercepted.

This is a problem for a reflective implementation because all invocations should be intercepted. Whether the semantics of invocation are redefined is being implemented at the metalevel. For example, a metaobject implementing access

\(^3\) In Java the self variable is represented by the this keyword.
control might need to redefine the semantics of only those invocations that come from clients of the class not from the class itself whereas a metaobject implementing resource controls would need to intercept and control every invocation regardless of source.

The second problem, an encapsulation problem, is based on the fact that the wrapping implemented above is only a "logical" form of wrapping. The unwrapped class is normally only hidden through a change of name. If the new name is known then it can be used by a programmer and this would allow access to the unwrapped class and new instances of it to be constructed. Another encapsulation problem arises if a method in the new class returns a pointer to the wrapped class. Once a client receives this pointer it can bypass the wrapping by sending method invocations directly to the wrapped class, bypassing the new class that logically wraps it. In order to solve these two problems Holze proposes removing the separate class that acts as the logical wrapper by unifying the wrapper and object at binary level. If there is no separation then self will refer to the wrapper and the object, and there is no way to break encapsulation. We will discuss how this approach relates to the implementation of reflection in Java in section 6.

Fig. 3. Interaction diagrams illustrating what happens when a method invocation is sent to WrapperA. In the delegation scenario WrapperA receives method1 invocation and invokes method1 on A. A then invokes method2 on itself which as self is still bound to WrapperA results in the invocation going to WrapperA where it is then delegated to A. In the forwarding scenario WrapperA receives method1 and forwards the invocations to A. A then invokes method2 and since self is rebound to A it is invoked on A directly bypassing WrapperA.

4.2 Inheritance

There are two different aspects of inheritance that cause us concern. The first relates to the type hierarchy imposed by the use of a class wrapper, and the second relates to inheritance of metaobject classes. We feel that such implications of inheritance are often neglected in implementations of reflective languages,
with the notable exception of CLOS [20] and SOM [9]. We discuss each problem below. The first problem is that the class wrapper that we create to logically wrap the base class does not share the same type hierarchy as the baselevel class. Instead it extends the metalevel type hierarchy in order to inherit the metaobject implementation. In order that it can be used interchangeably with the baselevel class the interface of the baselevel class (including its superclasses) is replicated in the new class and the class must be tagged as implementing all the interfaces that the baselevel class implemented. Otherwise the new class wrapping the baselevel could not be downcast to any of the interfaces. However, any downcast to one of the supertypes of the baselevel class will fail as, although the interface is present, Java does not see that the class wrapper is a subtype. This could lead to problems with existing client code. Ideally the type hierarchy of the class wrapper should reflect the baselevel application level type hierarchy.

The second problem is the well known meta constraint problem [9]. If a class that is bound to a metaobject class is extended by another class and that class is bound to a different metaobject class then there should be some way of sensibly resolving the effect of both metaobject classes on the resultant class. In Dalang the methods inherited from the superclass are controlled by both metaobject classes, whilst those methods introduced by the extending class are only controlled by the later metaobject. In contrast, in SOM a new metaclass would be synthesised that solved the constraints on the two metaobject classes and this new metaclass would control the methods of the extended class. The Dalang result appears inconsistent as we would expect on a logical level that if metaobject classes are thought of as adjectives then extending a red plane to make a jetplane and then applying the adjective fast should result in red and fast being applied to every method of the jetplane not just the methods inherited from plane. Alternatively only the last applied metaobject class should control the behaviour of the methods of the extended class. An in-between situation seems unsatisfactory.

4.3 Security Considerations of Applying Metaobjects

How do we know that a base class is not bound to a malicious metaobject? This could occur through either the metaobject binding specification being changed, or the metaobject itself being substituted with a malicious version.

In the current Dalang security model we assume that both the metaobject classes and the binding specification represented by a metaconfiguration file are trusted. However, if the metaobject has been replaced or corrupted then this introduces a major vulnerability into the system as the malicious metaobject could completely redefine or interfere with the normal operation of the application object it is bound to.

Ideally the JDK1.2 security infrastructure for securing ordinary classes should be also used for metaobject classes. This supports the digital signing of classes using private keys and the checking of the signatures at loadtime by a SecureClassLoader. When a class is loaded it has its signature checked automatically and fine-grained controls are placed on its access to system resources.
The ReflectiveClassLoader should extend the SecureClassLoader class and throw an exception if the class is not signed with the expected key.

Similarly, some form of protection must be implemented for the metaconfiguration file. Again a digital signature would be appropriate for the integrity check, and the structure of the file should include ownership and change information. If the metaconfiguration file fails its integrity check a runtime exception should be thrown. These two countermeasures will ensure that metaobjects and metaconfiguration file can be trusted (assuming that the Java platform, and Dalang implementation have not been corrupted - however, the Dalang implementation can also be secured using a SecureClassLoader).

4.4 Class loaders

Several authors [2][8][19] make use of a user-defined class loader to support their extensions to Java as this allows transparent interception and redefinition of class loading. The class loader is used to bootstrap the application which means that the class loader will load all classes referenced in the first class it loads. Therefore it can apply appropriate transformations to all classes. However, if another user-defined class loader is loaded then any class loaded by the new class loader will bypass the bootstrapping class loader. In the case of Dalang this means that these classes cannot be made reflective.

This could be avoided if we changed the Java platform either by modifying the standard class loader or by changing the native code that instantiates a class. However, this would reduce the portability of Dalang and would not make sense if a change was required for every extension to Java. Arguably the JVM should provide some kind of facility for injecting before and after wrappers around the primordial classloader. As the result of class transformations are always verified by the JVM and don’t bypass type checks then this should be a safe approach.

5 Transparency and Reflection

A number of authors, including ourselves, have argued that reflection can be used to transparently implement non-functional requirements (NFRs) such as security [8], fault tolerance [11], distribution [29], atomicity [31], and concurrency [33]. The idea is that components or classes developed independently can be modified in a principled way in order to support new properties that are orthogonal to their functionality. The approach provides reusable implementations of NFRs. Our experiences with Dalang suggest that there are the following problems with a completely transparent application of reflection:

1. Recursive Reflection
2. Exceptions
3. Composability

There is a related problem with composing class loaders using the JDK1.2 delegation model but there are no current plans to fix it [3]
The problems discussed in this section are not handled by the current Dalang metaobject protocol but have influenced the design of its successor Kava.

5.1 Recursive Reflection

In Dalang the reflective class loader loads all classes referenced by a reflective class. This means that if necessary these classes can be made reflective. Currently a referenced class will not be made reflective unless specifically bound to a metaobject class in the metaconfiguration file. However, what if the binding is not known ahead of time and the real requirement is that the binding between the initial class and metaclass be recursive in applicability? Thus if the initial reflective class references another class, that class should also be bound to the same metaclass. An example would be persistence, should all classes referenced by a root class be made persistent and a deep copy made when the class is stored? A counterexample might be Java-style serialisation, where it is not appropriate to make a deep copy and a shallow copy only is required. An example of this problem can be seen in the FRIENDS scheme proposal for tracing state changes for checkpointing using reflection [21]. Which changes of state should be tracked is application dependent - in some cases the object referenced by the object being checkpointed should also be checkpointed but in other cases it is a volatile object that should not be checkpointed. There must be some form of application control over recursive reflection at the metalevel either through some policy file such as metaconfiguration file or a metalevel program.

5.2 Exception Handling

Exceptions are an integral part of most modern object-oriented languages. However, it seems to us that they are rarely explicitly addressed by designers of reflective languages with the exceptions of the Lore and Smalltalk languages [10] where exceptions are implemented reflectively. Recently, our colleagues at York University [4] have argued that using reflection to implement exceptions allows adaptive runtime redefinition of exceptional behaviour. We argue that the behaviour of exception handling should not be redefinable at runtime as it makes validation intractable but nonetheless there is a need to deal with exceptions in the metalevel. Exceptions should be viewed as another possible outcome of a method call. If the behaviour of method calls is reflected upon, then in the same way as the return value of a method call is reified and available to the metalevel, so the raising of an exception by a method should be as well. This need finds expression in two ways. The first is what control and representation is appropriate for exceptions generated at the application level in the metalevel? The second is how do we deal sensibly with exceptions that occur at the metalevel that were never considered when the reflected component was constructed?

Firstly, although we do not want to redefine exception handling there are situations where being able to detect the raising of an exception at the application level, and to switch to the metalevel, is desirable. Consider the intention of taking components and distributing them through the use of reflection. Co-operating
components may need to participate in a distributed exception handling routine. This requires the interception of exception raising and switching to the metalevel to propagate the exception to associated distributed components before resuming.

Secondly, when using reflection to add a non-functional characteristic new types of failure become possible that couldn’t occur previously. For example, if a component is bound to a client metaobject that allows distributed invocation then a failure such as the loss of a method invocation will generate an exception at the metalevel that the component at the application level was never designed to raise. One approach to this problem would be to modify the application level component on the fly to support the new exception types that can be raised in the metalevel. However, this simply leads to propagation of the problem as any client of the component would also need to be able to handle the new exceptions. Perhaps the only sensible approach is to mask the exception and raise a runtime exception and hope that the existing application handles runtime exceptions in a sensible way. This is not an ideal solution but appears to be a necessary consequence of trying to reuse transparently an independently developed metalayer in a transparent way with an independently developed application layer.

5.3 Composability

In the same way that independently developed metaobject classes are combined with independently developed application components, independently developed metaobject classes may be combined with each other. With wrapper based approaches each object has one statically bound metaobject wrapper. However, this does not mean that only one behaviour can be enforced on an object. This is because chaining can be used to compose metaobjects together at the same metalevel to create a complex meta architecture (similar to the approach described in Mulet et al. [27], or Oliva [28]). We believe that using chaining to achieve composition of metaobjects is more appropriate than using reflection recursively to build a tower of metaobjects (the approach adopted by FRIENDS [11]). The advantage of using chaining is that each metaobject may introspect on the baselevel object whereas with a metaobject tower only the base metaobject may introspect on the baselevel object. In addition, with chaining the cost of interception and reflection of method calls is only incurred once as we do not have to transform the metaobject classes involved in the tower. The main problem with the tower approach is that it confuses the meta interception mechanism with the metalevel structuring mechanism, which we feel should be kept separate. However, the tower approach is appropriate only when going to a new level of abstraction such as in the ABCL/R [24] approach.

6 Byte Code Transformation

In this section we introduce a new metaobject protocol based on byte code transformation that we are implementing as the Kawa system. As stated in section four
the separation of wrapper and wrapped class introduces problems for method call interception, encapsulation and typing. The solution as proposed by Holzle [16] is to unify the wrapper and wrapped class at the binary level. In order to achieve this at load time in Java we need to transform classes at the byte code level using byte code transformation techniques. In this section we discuss the general approach, introduce the two types of standard transformations used to implement MLIs, provide an example of the application of a standard transformation, highlight which aspects of the Java object model can be changed and examine some tricky issues.

6.1 General Approach

Byte code transformation has become an established technique for extending Java both syntactically and behaviourally. For example it has been used to support parametric types [2] and add resource consumption controls to classes [8]. Generic frameworks for transforming byte code such as JOIE [7] and Binary Component Adaptation [19] have been developed to make coding byte code transformations easier. However, as pointed out by the authors of JOIE, most of these frameworks lack a high-level abstraction for describing the behavioural adaptations. This makes coding adaptations difficult as it requires a detailed knowledge of byte code instructions and of the structure of class files. Binary Component Adaptation does support a form of a higher-level abstraction in that it has the concept of deltaClasses that describe structural changes to a class file in terms of methods for renaming methods, mixin type methods, etc. However, the purpose of the framework is to support software evolution rather than behavioural adaptation. This means that the focus is on adding, renaming or removing methods and manipulating the type hierarchy in order to adapt ill-fitting components to work together rather than describing behavioural adaptation.

Our contribution is to provide a high-level abstraction for adaptation of component behaviour that specifies the change to behaviour in terms of the Java object model and is implemented using byte code transformation. We exploit the JOIE framework to simplify the implementation of this metaobject protocol. The framework frees us from dealing with technical details such as maintaining relative addressing when new byte codes are inserted into a method, or determining the number of arguments a method supports before it has been instantiated as part of a class.

As byte code instructions and the structure of the class file preserve most of the semantics of the source code we can use byte code transformation to implement MLIs for a wide range of aspects of the Java Object model such as caller and receiver method invocation, state access, object finalisation, object initialisation and some aspects of exception handling. Like compile-time reflection we reflect upon the structure of the code in order to implement reflection. However, we work at a level much closer to the Java machine than most compile-time approaches which deal with the higher-level language. Although this means we cannot extend the syntax of the higher-level language it does mean that we can
implement some kinds of reflection more easily than in a traditional compile-time MOP. For example, in the application of OpenC++ version 2 to adding fault tolerance in the form of checkpointing CORBA applications [21] data flow analysis is performed on the source code to determine when the state of the object is updated. With Kava no such analysis would be necessary; all that would be required is to intercept the update of state of an object by reflecting upon the behaviour of the update field byte code instruction. When an update was done a flag could be set indicating that the current state should be checkpointed.

By transforming the class itself we also address the problems introduced by the separation of baseclass and class wrapper. Instead, standard transformations of byte code are used to wrap individual methods and even byte code instructions. These micro-wrappers will switch control from the baselevel to metalevel when the methods or byte code instructions are executed at runtime. As with Dalang the metalevel will be programmed using standard Java classes. The metalevel will allow the customisation of the Java object model at runtime. The scope of the customisation will be determined by which methods and byte code instructions are wrapped at load time, but the exact nature of the customisation will be adjustable at runtime.

Figure 4 shows the class collaboration diagram for Kava. Note that there is no separate class wrapper. This is because wrappers have been applied within the base level class. Whenever the BaseObject has a method invoked or it performs state access the appropriate method of the bound ConcreteMetaObject is invoked before and after the operation. For example, when a method invocation is received the associated ConcreteMetaObject's method beforeReceiveMethod() is called with parameters representing the source, arguments, and method. Subsequently, when the invocation takes place the method afterReceiveMethod() is called with a parameter representing the result of the invocation. The ConcreteMetaObject can cooperate with any number of other ConcreteMetaObjects to implement the overall behavioural adaptation. All ConcreteMetaObjects extend the abstract MetaObject which defines the default functionality for a MetaObject.

6.2 Standard Transformations

There are two types of standard transformation applied by Kava to add MLIs to classes.

The first standard transformation makes use of the structure of a class file to identify blocks of byte code representing class methods, initialisation methods, and finalisation methods and then adds wrappers around them. This allows the method invocations sent to the base level to be intercepted and control handed to a metaclass. This is similar to the type of MLIs implemented in Dalang. In order to make the metaobject aware of self directed invocations all self invocations in the base level class are rewritten to pass an extra parameter indicating the source of the invocation.

The second standard transformation is applied to byte code instructions such as those dealing with invocation, access to state and wraps these individual in-
structions. Here fine-grained wrappers are applied around individual instructions. This allows the interception and switch to the metalayer when the class itself makes an invocation or accesses state. These types of interceptions would not be possible if byte code transformation had not been implemented.

In order to safely insert new byte code instructions into a class, some difficult technical issues must be solved. For example, how to handle the effects of inserting instructions on relative branches on other branch instructions and the exception table. Fortunately, we can rely upon the **JOIE** framework to take care of these low-level issues.

![Class collaboration diagram for Kava. Shows the relationship between classes.](image)

### 6.3 Example

To provide a flavour of this approach we provide an example of the wrapping of access to a field of a base level class. Due to space constraints we present this at a high level using source code instead of byte code. Consider the following field access:

```java
helloWorld = "Hello " + name;
```

At the byte code level this is rewritten to:

```java
Value
fieldValue = Value.newValue("Hello " + name);
meta.beforePutField("helloWorld", fieldValue); try { helloWorld = (String)fieldValue.getObject(); } catch (Meta SuppressUpdateException e) { } meta.afterPutField("helloWorld", fieldValue);
```
In this example the first line of code marshalls the value that the field is going to be updated with and then, the beforePutField method of the associated metaobject is invoked with parameters representing the name of the field ("helloWorld"), and the marshalled value.

At the metalevel the value may be modified in order to adjust the final value that is stored in the field. Alternatively the update of field could be suppressed by the throwing of a MetaSuppressUpdateException. This is caught at the base level and causes the update of the base level field not to take place.

The last line calls the afterPutField method of the associated metaobject with the same parameters as the initial call to the metalevel.

### 6.4 Reflective Aspects of Java Object Model

In this section we highlight the aspects of the Java object model that can be reflected upon by Kava.

As object-oriented operations such as object creation, state access and invocation are represented directly as byte codes it is relatively straightforward to identify them within class methods. Once identified the JOIE framework can be used to determine the arguments for these operations which in turn allows us to dynamically construct bytecode that marshalls their values. We can then place before and after calls around the individual operations that can manipulate the arguments and even suppress the operations. We term this caller side reflection because it allows us to capture behaviour such as the calling of another method by a class.

As the structure of the class file is known we can easily identify blocks of byte code representing method calls, object initialisation and finalisation methods. Again we can use the JOIE framework to determine the arguments for these methods and dynamically generate marshalling code. We can then insert calls to the metaobject’s before method appropriate to the type of method being instrumented at the beginning of the method's byte code instruction block. Then we insert calls to the metaobject’s appropriate after method before every return instruction in the method. This allows us to intercept received invocations and represents what we term receiver side reflection.

Note that with both caller side reflection and receiver side reflection we add MLIs into the class’ byte code. Each MLI added using byte code transformation switches control from the baselevel to the metalevel which is implemented using metaobject. This means that at runtime we can perform dynamic reflection to adjust the runtime behaviour of the baselevel class by either changing the implementation of the before and after methods of the metaobject associated with the baselevel object or by changing the binding to another metaobject. With such a facility we can easily support per-class instance metaobjects or per-instance metaobjects.

Once a class has been instantiated in the JVM no more MLIs can be added to it. This means that if a MLI has not been added at load time it cannot be exploited at any later stage. For example, if a MLI is not added to a method of a class at load time that method cannot be brought under the control of a
metaobject at a later stage. One way of addressing this problem would be to add all possible MLIs by default. However, this would have considerable performance considerations. Ideally we would wish to see some form of lazy reflection taking place with MLIs being switchable on and off once the class is executing in the JVM. There is some lazy evaluation that takes place in a standard JVM where byte codes are replaced at runtime if required and perhaps this could be exploited to provide a form of lazy reflection where a class can be made reflective only as required. Another approach would be to use an OpenJIT to add interceptions as required when the Java byte code is compiled to native machine code.

7 Current Work

We are currently implementing the Kava system. Kava provides the portability benefits of a class wrapper approach without many of the problems. It also provides the opportunity to address some of the more general problems faced by reflective languages. Kava will have the following characteristics:

Encapsulation. The MLIs will be non-bypassable as class wrapper and base class are no longer separate. This is important for supporting applications such as security.

Self Problem. It will be possible to specify at the metalayer whether self rebinds or not as this is dependent on the metalevel functionality being provided.

Reflective Capabilities. Unification will allow both reification and reflection upon the receipt of method invocations and also the sending of method invocations. In addition, reification and reflection upon construction and finalisation will be possible. State access will also be under control of the metalayer. There will be some representation of exception handling at the metalayer.

Transparency and Security. We envisage a security model such that a trusted Kava kernel exists on the target machine and secure class loaders are used to download application code, metaobject code and the metaobject binding configuration. The use of a secure class loader is necessary so that we can trust the identity of the code and check for tampering. Alternatively where the client machine is totally untrusted we propose that Kava be applied before delivery of code to the client either in a web server or through a proxy server.

Inheritance. Due to unification the wrapper and wrapped class will have identical superclasses and the same implementations of application methods. This means that the class wrapper will be indistinguishable from the baselevel wrapped class. Also a subclass of the class wrapper will also inherit any reflective behaviour.

Metaclass constraints. Given that we can construct classes on-the-fly at runtime it will be possible to implement metaclass constraint solving [9].

Recursive Reflection. Like the self problem, whether reflection is applied recursively should be under the control of the metalayer, because the correct behaviour depends on the functionality provided by the metalayer. The recursive class loader should support a metaobject protocol that allows the metalevel pro-
grammer to customise its behaviour and devise sophisticated policies for handling recursive reflection.

**Metaclass Combination.** *Kava* will provide default metaobject classes that support metaclass combination and dynamic rebinding of metaobjects at runtime. We will provide metaobject classes that can be subclassed from that support co-operation between metaobjects, and also the ability to dynamically switch the binding between base and metalevel.

**Exception Handling.** Although we cannot redefine how the JVM handles the throwing of exceptions we can provide the ability to detect the throwing of exceptions and switch to the metalayer. This should provide the ability to support features such as distributed exception handling.

**Visibility of the Metalevel.** One aspect that should be configurable by the metalevel programmer is the visibility of the metalevel. Certainly in some cases we want to be able to send method invocations to the metalevel in order to tweak an aspect of the behaviour of the metaobject. For example, we suggest that timeout values of a metaobject that handles distributed communication could be controlled by sending messages to the metalevel.

8 Example Applications of *Kava*

*Kava* has wide application potential and should prove well suited to the customising the behaviour of COTS applications that are built from dynamically loaded components. As it provides a high-level abstraction for implementing bytecode transformation it could be used to implement behavioural adaptations that have already been implemented through byte code transformation e.g. enforcing resource controls on applications, instrumentation of applications, visualisation support etc. The advantage of using *Kava* would be that these adaptations could be combined as required since they have been built using a common abstraction.

9 Conclusions

Ideally, a reflective extension for Java that is intended to be used to adapt the runtime behaviour of COTS components should not require access to source code, or modifications to the Java platform. Unfortunately, the reflective extensions that we have reviewed do not meet these requirements.

Using class wrappers appears a promising way to implement a reflective extension that does meet the requirements. However, as our experience with *Dalang* has shown there are serious drawbacks to implementing reflection using this general approach. Aside from these problems there are other general issues that must be addressed when attempting to apply reflection transparently to COTS software. Some of the problems identified and discussed were problems associated with inheritance, encapsulation, exception handling etc.

Having identified these problems we discussed how we plan to address a number of these issues by applying reflection to bytecode transformation techniques
in a new version of Dalang called Kava. Although neither reflection or byte code transformation are new concepts, what is new is the implementation of metaobject protocols using byte code transformation in order to provide a high-level abstraction for controlling component adaptation.

We are currently working on the implementation of Kava, and intend to use it to implement a reflective security metalayer.

10 Acknowledgements

We would like to thank DERA for their financial support for this work, Brian Randell for helping review an initial version of this paper, and Jim Whitman for his reviewing of the final version of the paper.

References


[36] Zimmerman, Chris: Metalevels, MOPs and What all the Fuzz is All about. Chapter 1 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.