USENIX Association

Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

San Antonio, Texas, USA
January 29 - February 2, 2001

Kava - Using Byte code Rewriting to add Behavioural Reflection to Java

Ian Welch and Robert J. Stroud
Department of Computing
University of Newcastle upon Tyne

Abstract

Many authors have proposed using byte code rewriting as a way of adapting or extending the behaviour of Java classes. There are toolkits available that simplify this process and raise the level of abstraction above byte code. However, to the best of our knowledge, none of these toolkits provide a complete model of behavioural reflection for Java. In this paper, we describe how we have used load-time byte code rewriting techniques to construct a run-time metaobject protocol for Java that can be used to adapt and customise the behaviour of Java classes in a more flexible and abstract way. Apart from providing a better semantic basis for byte code rewriting techniques, our approach also has the advantage over other reflective Java implementations that it doesn't require a modified compiler or JVM, can operate on byte code rather than source code and cannot be bypassed. In this paper we describe the implementation of Kava, our reflective implementation of Java, and discuss some of the linguistic issues and technical challenges involved in implementing such a tool on top of a standard JVM. Kava is available from http://www.cs.ncl.ac.uk/research/dependability/reflection.

1. Introduction

Many authors have considered the problem of reusing third party code in environments the developers did not originally consider [1, 2, 3]. For example, some proposals suggest ways to apply access control policies to code that has been developed without any thought for security [3]. Wrapping was originally proposed as a technique to enable adaptation of the code but it suffers from a number of problems such as identity confusion, or the self problem [1] etc. A solution to the problem is transform the code at the binary level [4]. This has proved to be a practical technique in the context of Java because the Java byte code retains a large amount of semantic information.

A number of byte code rewriting tools have been developed to ease the process of code rewriting. These include JOIE [5], Byte Code Engineering Library [6] and more recently Javassist [7]. Each toolkit provides object oriented representations of the structure of classes that can be used to rewrite classes on-the-fly.

The focus of these toolkits is on implementing changes to the behaviour of classes through programs that rewrite the class implementations. Users typically have to write programs that walk class structures and locate the appropriate places to make changes to the structure in order to implement some change to runtime behaviour. The actual implementation of changes this way is difficult for most programmers and highly error prone.

We argue that for applications where non-functional concerns are being implemented (security, transactions, debugging etc.) it would be more natural to specify changes to the behaviour of classes in terms of runtime abstractions.

For example, in order to trace state changes it would be more natural to redefine the runtime state access operation rather than manually write a program that walks all the methods of a class file and instruments pertinent field access operations.

Metaobject protocols and reflection are a good model for expressing such changes. Metaobject protocols provide abstractions of the runtime environment, and expose the protocols governing the execution in the runtime environment. Reflection means that changes to the implementation of these metaobject protocols will change the way in which code is executed at runtime.

We have implemented a highly portable implementation of a behavioural reflection for Java called Kava [8]. It provides a metaobject protocol for specifying changes to runtime behaviour and implements these changes through the use of structural rewriting toolkits such as JOIE, Byte Code Engineering Library, or Javassist. It is portable, is written entirely in Java, and unlike a number of other reflective Java implementations doesn't require a specialised Java Virtual Machine. Kava also provides support for properties such as strong non-bypassability and

```
public class TraceMethod implements Constants {
  private static String          class_name;
  private static ConstantPoolGen cp;
  private static int             out;     // reference to System.out
  private static int             println; // reference to PrintStream.println

  private static Method traceMethod(Method m) {
    Code   code  = m.getCode();
    int    flags = m.getAccessFlags();
    String name  = m.getName();

    // Create instruction list to be inserted at method start.
    String mesg = "tracing " + m.getMethodName();
    InstructionList patch  = new InstructionList();
    patch.append(new GETSTATIC(out));
    patch.append(new PUSH(cp, mesg));
    patch.append(new INVOKEVIRTUAL(println));

    MethodGen           mg  = new MethodGen(m, class_name, cp);
    InstructionList     il  = mg.getInstructionList();
    InstructionHandle[] ihs = il.getInstructionHandles();

    // First let the super or other constructor be called
    if(name.equals("<init>")) {
      for(int j=1; j < ihs.length; j++) {
        if(ihs[j].getInstruction() instanceof INVOKESPECIAL) {
          il.append(ihs[j], patch);  // Should check: method name == "<init>"
          break;
        }
      }
    }
    else
      il.insert(ihs[0], patch);

    // update stack size
    if(code.getMaxStack() < 2)
        mg.setMaxStack(2);

    return mg.getMethod();
  }

  public static void main(String[] argv) {
    JavaClass       java_class = new ClassParser(argv[i]).parse();
    ConstantPool    constants  = java_class.getConstantPool();
    cp = new ConstantPoolGen(constants);
    out     = cp.addFieldref("java.lang.System", "out",
                             "Ljava/io/PrintStream;");
    println = cp.addMethodref("java.io.PrintStream",
                              "println",
                              "(Ljava/lang/String;)V");

    Method[] methods = java_class.getMethods();
    for(int j=0; j < methods.length; j++)
      methods[j] = traceMethod(methods[j]);

    java_class.setConstantPool(cp.getFinalConstantPool());
    java_class.dump(class.getClassName()+".class");
  }
}
```

Figure 1 – Tracing Method Execution

reflection on inherited methods that other reflective Java implementations do not address.

In section 2 we discuss byte code rewriting and its shortcomings, in section 3 we introduce the Kava system, in section 4 we provide some examples of its application, in section 5 we discuss the implementation of Kava, in section 6 we provide an overview of related work and finally in section 7 we give our conclusions and outline future work.

2. Bytecode Rewriting

There are three main toolkits for rewriting bytecodes: Joie, Byte Code Engineering Library and Javassist. The first two toolkits provide object oriented frameworks for writing programs that manipulate the structure of class files. They provide loadtime representations of elements of class files such as methods, types, instructions etc. Java programs can then be written that describe how class files can be rewritten as late as load time. The main drawback with this approach is that the programmer has to have a detailed understanding of both the structure of class files and Java virtual machine programming. As the authors of Joie have observed, this makes it difficult for programmers to write reliable and easily understandable transformer programs. Javassist attempts to address this problem by providing a metaobject protocol for the rewriting of byte codes. It allows a programmer to work at a more abstract level. However, it sacrifices some of the power of the other toolkits without gaining a high enough level of abstraction. Also, it still requires the programmer to think in terms of reprogramming an existing implementation.

Figure 1 shows how the Byte Code Engineering Library can be used to trace method execution of a class.

This code adds a print statement at the start of each method. The `traceMethod` method generates the appropriate byte code for a print statement. While the main method traverses the structure of the class to locate the appropriate place to insert the instructions and finally ensure that the stack size after insertion is correct.

This process is obviously difficult for novice programmers to learn and is error prone. It is difficult as the code to be inserted is developed by hand and the programmer must manually add the appropriate entries to the constant pool. It is error prone because there is no separate type checking available for the code to be inserted. In addition to writing the code to be inserted the code for performing the insertion also has to be written from scratch every time and issues such as ensuring that the stack size is maintained correctly have to be addressed by the programmer.

To address these concerns, two improvements are needed:

- The ability to write the behavioural modifications in Java, and to be able to compile and verify these modifications as you would a normal class.
- The ability to declaratively specify where the behavioural modifications should be applied.

Kava provides these improvements. Behavioural adaptations are implemented using metaobject classes that can be compiled and verified, and the application of the metaobjects is driven by a binding specification that uses a declarative binding language.

3. Using Kava

In this section we introduce the basic concepts of behavioural reflection, and describe how Kava is actually used.

3.1. Behavioural Reflection

Reflection [9] is the process by which a system can reason about and act upon itself. A reflective system is composed of a base level and a meta level. The base level is the system being reasoned about, and the meta level has access to representations of the base level. *Reification* is the process by which the abstract representations of the base level are generated. A reflective system has the property that the meta level is *causally connected* to the base level. This means that changes at the meta level cause changes to the behaviour of the base level.

These notions of reflection have been extended to include the concept of the *metaobject protocol* [10] where an abstraction of the computation process and the protocols governing the execution of the program are exposed. A *metaobject* is bound to an object and controls the execution of the object. By changing the implementation of the metaobject the object's execution can be adjusted in a principled way. The protocols are implemented as methods of the metaobject.

Reflection and metaobject protocols have been successfully used to implement non-functional properties such

```
public interface IMetaObject {

  public void beforeExecuteMethod(IExecutionContext context)
  public void afterExecuteMethod(IExecutionContext context);
   /* called when a method is executed (including constructor/finalizer) */

  public void beforePutField(IFieldContext context);
  public void afterPutField(IFieldContext context);
   /* called when a field is accessed */

  public void beforeGetField(IFieldContext context);
  public void afterGetField(IFieldContext context);
   /* called when a field is read */

  public void beforeInvoke(IInvocationContext context);
  public void afterInvoke(IInvocationContext context);
   /* called when a method is invoked (including initialser */

  public void beforeException(IExceptionContext context);
  public void afterException(IExceptionContext context);
  /* called when an exception is thrown and caught */

}
```

Figure 2 – Interface for Kava MetaObject

as concurrent programming [11], atomic data types [12], fault tolerance [13], and security [14].

The Java programming language [15] includes a reflection package. This provides the ability to reify some aspects of the Java runtime environment such as methods, classes, fields, etc. and allows dynamic construction of proxies and dynamic method invocation. However, it does not provide the ability to modify the behaviour of an application through changes at a meta level. Kava provides powerful behavioural reflection without requiring changes to the Java Virtual Machine or requiring the use of source code preprocessing. It implements behavioural reflection through the principled rewriting of Java class files.

The Kava system allows each object or class to be bound to a metaobject. At the meta level runtime behaviours such as method invocation, method execution, field access, etc. can be redefined by the metaobject implementation. The metaobject implementation is constructed using reified aspects of the runtime object model. For example, a method is reified as an instance of a `Method` class.

The binding itself is described by a binding specification. This is written using a declarative binding language. Separating the binding information from the metaobjects increases the reusability of metaobjects as the bindings effectively parameterise the metaobjects. For example, a binding specification may bind a metaobject to different fields on different classes.

3.2 Using Kava

Each metaobject is an implementation of the interface `IMetaObject`. This defines a series of methods for intercepting and customising various aspects of the runtime behaviour of an object. See Figure 2 for the interface.

Each method has a *before* and *after* variant. The before methods are invoked before the behaviour, and the after methods are invoked after the behaviour. Each time a metaobject's method is invoked the behaviour's context is reified as an instance of a context object and passed as an argument. This makes the context accessible to the metaobject implementation. Some aspects of the context can be changed at the metalevel, such as the actual arguments passed to a method. On return to the base level the context object is converted back to the actual context of the behaviour.

Each before method can set the context such that the base level behaviour is overriden. This means that the base level behaviour will be suppressed. For example setting an override in a `beforeExecuteMethod` will result in the body of the method not being executed.

An example of a metaobject that implements the tracing of method executions similar to the example given in section 2 is:

```
public class MetaTrace
             implements IMetaObject {
  public void
     beforeExecuteMethod(
       IExecutionContext context) {
         System.out.println(
            "tracing " +
             context.getMethodName());
  }
}
```

In order to trace the methods of a particular class, it is necessary to establish a binding between instances of the class and instances of the `MetaTrace` class. These bindings are described using the Kava binding language in a special metaconfiguration file that drives the processing of a class by Kava. The binding specification shown below means that `MetaTrace` intercepts the execution of any method of the class `Test`.

```
<binding>
  <class>
    <classname>Test</classname>
    <metaclass>MetaTrace</metaclass>
    <intercept>
      <execute>
        <method>*</method>
        <parameters>*</parameters>
      </execute>
    </intercept>
   </class>
</binding>
```

If the implementation of `Test` is:

```
public class Test {
  public static
    void main(String[] args){
      (new Test).run(args[0]);
    }
  public void run(String s) {
      System.out.println("hello " + s);
  }
}
```

Then output of invoking the `run` method of `Test` with the actual parameter `World` is:

```
tracing run
hello World
```

Note that the code necessary to implement tracing behaviour is significantly more concise than the equivalent byte code transformation code. The metaobject that specifies the code to be invoked when a method is executed can also be compiled and verified therefore reducing the possibility of coding errors. The binding specification is significantly shorter than the code that traverses the class and inserts instructions at the appropriate place. Also, since it is a declarative specification it is easier to code and less likely to contain errors.

Kava is well suited to modifying the behaviour of classes where the interface of the class is not to be changed, or new keywords to be added to the language. As this example shows it is far more concise than an equivalent byte code transformation program, and it separates out the adaptation code (the metaobject) and the specification of where to apply the adaptation (the binding).

4. Examples

This section shows applications of Kava that highlight some of the more unusual features of the Kava metaobject protocol. Many implementations of reflective Java concentrate on intercepting method calls and tracing method calls is the standard example used to demonstrate a reflective system. Kava provides the ability to intercept the sending of method calls (invocation), field access, and exception handling in addition to the interception of method calls. The first example given here is of fine grained access control, this illustrates Kava's ability to control field access. The second example given here is how to prevent a particular type of denial of service attack, this illustrates Kava's ability to intercept the sending of method calls.

4.1 Fine grained access control

The Java programming language provides the following language level mechanisms for controlling access to class members such as methods or fields:

- Public access where code belonging to any class is allowed to access the member.
- Package access where access to the member is permitted only to code belonging to classes in the same package.
- Protected access where access to the member is permitted only to code that inherits from the functionality of the class.
- Private access where access to the member is permitted only to code that occurs in the body of the top level class that encloses the declaration of the member.

While this is adequate for a number of situations there is still the possibility that a more fine grained access control may be required for security purposes. For example, we may only want a certain field to be accessed by a limited number of classes that are spread across multiple packages.

Using Kava it is relatively simple to implement such a fine-grained scheme. In this example we focus on preventing access to fields by any but a small number of classes.

We implement the following protection metaobject `MetaChkAccess` that restricts access to a field to instances of two known classes `GoodGuyA` and `GoodGuyB`:

```
public MetaChkAccess implements
   IMetaObject {

  public void
    beforePutField(IFieldContext c) {
      checkAccess(c.getBase());
    }
    /* check any writes to the field */

  public void
    beforeGetField(IFieldContext c) {
      checkAccess(c.getBase ());
    }
    /* check any reads from the field */

  public void
    checkAccess(Object who) {
      if (who instanceof GoodGuy1 ||
        who instanceof GoodGuy2) {
        return;
      }
    else {
      // wrong class
      throw new
        SecurityException(
          "illegal access by  " +
          who.getClass().getName());
    }
  }
  /* check whether access is allowed */
}
```

The `checkAccess` method checks any access to a field. If a class other than one of the allowed classes attempts to access a field then a `SecurityException` is thrown. As `SecurityException` is a subclass of `RuntimeException` it does not have to be included in the declaration of the method.

The `MetaChkAccess` metaobject is then bound to any class that reads from or writes to the field `protectedField` of the class `ProtectedClass` by including the following in the binding specification:

```
<binding>
  <class>
    <classname>*</classname>
    <metaclass>MetaChkAccess</metaclass>
    <intercept>
      <getfield>
        <class>ProtectedClass</class>
        <field>ProtectedField</field>
      </getfield>
      <putfield>
        <class>ProtectedClass</class>
        <field>ProtectedField</field>
      </putfield>
    </intercept>
   </class>
</binding>
```

4.2 Denial of Service

Denial of service attacks are trivial to implement in Java. The simplest attacks consume resources by generating infinite numbers of objects such as windows that fill up the user's screen and occupy CPU time. Trivially this could be dealt with by defining a `MetaRsrceLmt` that watches how many instances of windows (all subclasses of `java.awt.Frame`) are created and limiting the number that can be created to a maximum:

```
public MetaRsrceLmt implements
  IMetaObject
{
  public void
    beforeInvoke(IInvocationContext c)
  {
    if (c.getTarget() instanceof
         java.awt.Window &&
        c.getMethod ().equals("<init>"))
    {
       maxCount++;
       if (maxCount > ARBITARY_MAX)
       {
          throw new
             RuntimeException(
     "exceeded max number of frames");
       }
    }
   }
}
```

This metaobject then would be bound to all method invocations by any method of any class:

```
<binding>
  <class>
    <classname>*</classname>
    <metaclass>MetaRsrceLmt</metaclass>
    <intercept>
      <invoke>
        <method>*</method>
        <parameters>*</parameters>
        <class>*</class>
        <targetmethod>
          <init/>
        </targetmethod>
      </invoke>
    </intercept>
   </class>
</binding>
```

A more sophisticated metaobject will allow the blocking of windows until one was destroyed, and would maintain a global count of windows and detect when windows were destroyed as well as created. However,

this example shows that resource creation can be easily controlled using a reflective approach.

5. Kava Implementation

5.1 Architecture

Kava is written purely in Java, it does not require any special Java Virtual Machine to work. The link between metaobjects and objects is realised by the rewriting of classes and addition of hooks into the class code. Figure 3 shows the Kava architecture. A classloader reads the class file as a stream of bytes. These can be retrieved from any source, normally from a file or from across the network. The classloader parses the byte stream and creates a JVM specific representation of a class. Normally this is passed to the verifier before it is instantiated by the JVM. However, *Kava* is used to intercept the byte stream before the classloader constructs the JVM specific class and applies the standard code transformations that realise control by metaobjects. As stated earlier Kava uses a binding specification file to determine what behaviours of what classes are to be brought under the control of particular metaobjects. It then adds traps into the code of the class to switch control when from the base level to the meta level (the associated metaobject) when the byte code base level objects carry out certain behaviours. After rewriting the class to include these traps, the classloader passes an internal representation of the class to the byte code verifier as before. This means that properties such as type safety are still honoured as before.

Note that the metaobjects are loaded by the classloader in exactly the same way as any other class, which means that they must satisfy the same security properties as any ordinary Java class. Metaobjects are ordinary Java classes and can be compiled which means that errors can be caught at an early stage.

Kava can be invoked either after a class is compiled or at the time the class is loaded into the JVM. In order to invoke Kava at loadtime a user-defined classloader must be used. In either case the traps that are added to the class are non-bypassable.

5.2 Instrumentation

The Kava metaobject protocol is implemented using the technique of byte code rewriting. Kava makes use of the Byte Code Engineering Library [6] toolkit to implement the standard transformations that add the hooks necessary to switch control from the base level to the meta level at runtime. Using a standard byte code rewriting toolkit 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.

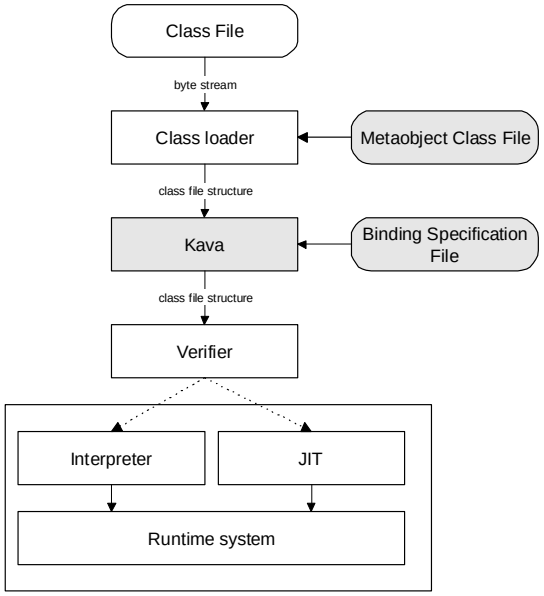

Figure 3 - Kava Architecture

Standard byte code rewritings are used to add hooks for individual methods and individual byte code instructions. These hooks reify the context of a behaviour that is being trapped, invoke the metaobject associated with an object and reflect any changes to the context back to the base level. The metaobjects that are invoked are completely separate from the byte code hooks and are developed entirely in Java. This separation means that the runtime meta level can be adjusted dynamically at runtime although which behaviours are trapped is determined at loadtime.

For example, returning to the example of section 3.2 the class `Test` included the following `run` method:

```
public void run(String s) {
  System.out.println("hello " + s); }
```

After the metaobject `MetaTrace` is bound to `Test` using the binding specification presented earlier, the `run` method is effectively rewritten by Kava as:

```
public void run(String s) {
  Context c = new Context
    (this, "run", "void",
     "java.lang.String", new Object[]
     {s});
  getMeta().beforeExecuteMethod(c);
  if (!c.override()) {
  System.out.println("hello " +
     (String)c.getArg(0));
```

```
    getMeta().afterExecuteMethod(c);
  }
}
```

The code in **bold** has been added by Kava.

First, at the beginning of the method block a context object that represents the invocation frame is created. It contains a pointer to the base level object itself, the name of the method being executed, the return type, the types of the parameters, and the actual parameters marshalled into an array of objects. Then the metaobject associated with the base level object is retrieved using a method added earlier by Kava and the `beforeExecuteMethod` method invoked. In this case `getMeta()` returns a pointer to an instance of a `MetaTrace` so the method name is printed out. Following the invocation of `beforeExecuteMethod` the arguments passed within the context object are unpacked, and the base level code is invoked. Finally, at the end of the method block the `afterExecuteMethod` method of the associated metaobject is invoked. In this case there was no implementation of the method so nothing occurs at the meta level.

Since a metaobject method may override the corresponding base level behaviour we add an if ... then clause. This ensures that when an override is indicated then the base level behaviour is suppressed.

This is an example of standard transformation for a block of code. The transformation for intercepting behaviour such as setting the value of a field is very similar but finer-grained with the hook code being around a single instruction.

5.3 Binding language

As explained in section 5.1 the binding specification file determines where Kava introduces the hooks into the base level code. The concept is that to make metaobjects more reusable the binding should be specified completely separately of both the base and meta level.

The binding specification contains multiple base object and metaobject class bindings. Each binding is between one class and a metaobject class. For that binding the particular behaviours to be brought under the control of the metaobject are specified, for example the execution of methods, or the setting of fields. These are parameterised by information such as the name of the field or method, the type of the target (in the case of setting a field, or invoking a method) etc.

5.4 Special Features

In this section we give an overview of some of the special features supported by Kava: strong encapsulation, reflection on inherited methods, exception handling and context objects.

5.4.1 Strong encapsulation

One of the benefits of the Kava implementation is its support for strong encapsulation. Strong encapsulation is the property that it is difficult to bypass the metaobject bound to the base level object. This has been achieved by avoiding the use of a separate wrapper class. Since hooks are added directly into method bodies we greatly reduce the possibility that the hooks could be bypassed. This is because there is no way to express in the Java language a branching to an arbitrary point in a method body.

It is true that if a malicious code transformer rewrote a class file that was pre-processed by Kava then our hooks could be removed. However, this can be easily guarded against through the use of a mixture of operating system protection and the use of code signing techniques.

5.4.2 Inherited Methods

When a Java class inherits a method from its superclass the bytecode implementing the method is not reproduced in the implementation of the class. For example, if class `C` inherits the `run` method from class `D` then the byte code for `run` is still to be found in `D`'s class file not `C`'s class file. If we bind `C` to a metaobject metaclass `MC`, and try and bring the execution of `run` under the control of `MC`, Kava will fail because it cannot find the byte code implementation of run. The obvious answer is to bind `MC` to `D` as well and add the hooks into `D`'s run method. However, we may not want `D` to be brought under the control of `MC`, indeed we may even want it to be bound to an entirely different metaclass.

The answer to this problem is to add a method `getMeta` to each base level class that returns a pointer to the metaobject bound to the base level object. We then ensure that superclass methods inherited by classes that are bound to a metaobject have hooks added to them that use the `getMeta` method to determine which metaobject to invoke.

When an instance of `C` has its inherited method `run` invoked the JVM's dynamic resolution of method calls will mean that the `getMeta` method appropriate to `C` will be invoked. This means that the metaobject bound to `C` will be returned.

When an instance of `D` had its method `run` invoked, the JVM's dynamic resolution of method calls will mean that the `getMeta` method appropriate to `D` will be invoked. This means that the metaobject bound to `D` will be returned.

This approach ensures that the correct metaobject is invoked in both cases. The approach taken here is similar to that found in [3].

5.4.3 Exception Handling

Kava allows the raising and throwing of exceptions to be intercepted and handled by the metaobject bound to an object. The `beforeException` method is invoked before an exception is thrown at the base level. It allows the exception throwing to be overridden, this might be necessary where the metaobject is implementing distribution at the meta level and the exception has to be propagated to a remote client. The `afterException` method is invoked after an exception has been thrown or raised at the base level. It doesn't allow overriding of this behaviour but does allow additional processing to take place such as the propagation of the exception to related objects if a number of objects are co-operating and need to be aware of each other's status.

5.5 Context Objects

Kava uses the concept of `Context` objects to simplify the metaobject protocol and also allow the possibility of lazy reification. The metaobject sometimes will need access to runtime instances of `Method`, `Class` or `Field`. However, generating these is a relatively expensive process so we defer their creation by passing the minimum information needed to derive them in a `Context` object. As part of the `Context` interface we provide methods for generating the reified instances. The standard Java reflective API is used to generate these instances. In the future we would like to apply the same technique to the actual parameters passed to the metaobject.

5.6 Performance

We have made some preliminary measurements of the performance of Kava. They indicate that the most expensive operation is the generation of the context. Presently, this expense is more than doubling the execution speed of a number of instructions. We are currently exploring two main approaches to improving performance. The first approach is to use caching of context information, and the second is to allow selective reification.

6. Related Work

In this section we briefly review a number of other reflective Java implementations and attempt to categorise them according to the point in the Java class lifecycle that reflection is implemented.

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.

Different reflective Java implementations introduce reflection at different points in the lifecycle. The point at which they introduce reflection tends to characterise the scope of their capabilities. In order to bring the base level under the control of the meta level the base level system is modified through the addition of traps. These traps are known as meta level interceptions [16]. For example, in Reflective Java method calls sent to the base object are brought under control of an associated metaobject by trapping each method call to the baseobject. This is done by pre-processing the source code of the base level class. A contrasting example is MetaXa where the traps are in the implementation of the dispatch mechanism of the Virtual Machine. As the traps exist in the Virtual Machine itself, the source code of classes to be made reflective is not required. However, unlike Reflective Java, a specialised JVM must be used.

Table 1 summarises the features of various reflective Java implementations. All these implementations have drawbacks that make them unsuitable for use with compiled components or in a standard Java environment where the purpose is to add security. Some require access to source code, and others are non-standard because they make use of a modified Java platform.

Point in Lifecycle	Reflective Java	Description	Capabilities	Restrictions
Source Code	Reflective Java [17]	Preprocessor.	Dynamic switching of metaobjects. Intercept method invocations.	Can't make a compiled class reflective, requires access to source code.
Compile Time	OpenJava [18]	Compile-time metaobject protocol.	Can intercept wide range of operations, and extends language syntax.	Requires access to source code.
Byte Code	Bean Extender [19], Dalang [20], JavaAssist [7]	Byte code preprocessor (Bean Extender), byte code rewriting as late as load time (Dalang, JavaAssist).	No need to have access to source code.	Bean Extender – restricted to Java Beans. Dalang, and Javassist – limited capabilities. requires offline preprocessing.
Runtime	MetaXa [21], Rjava [22], Guarana [23]	Reflective JVMs.	Can intercept wide range of operations. Can be dynamically applied.	Custom JVM.
	java.lang. reflect [15]	Reflective capabilities part of the standard Java development kit.	Runtime introspection, dynamic dispatch and on-the-fly generation of proxies.	Overall introspection rather than behavioural or structural reflection.
Just-in-time Compilation	OpenJIT [24]	Compile-time metaobject protocol for compilation to machine language.	Can take advantage of facilities present in the native platform. No need for access to source code. Dynamic adaptation.	Custom Just-in-time compiler.

Table 1 - Reflective Java Implementations

In contrast, Kava does not require access to source code because it is based on byte code rewriting, doesn't require a non-standard Java environment and provides a rich set of capabilities. It also provides what we refer to as *strong encapsulation*. Most implementations add traps through renaming of classes, or renaming methods, which means that it may be possible to call the original methods and therefore bypass the meta layer. Kava actually adds the traps directly into the method bodies avoiding this problem. Dalang was an earlier implementation of a loadtime reflective Java we implemented that suffered from this problem. See [20] for an account of the evolution of Kava from Dalang.

The closest reflective Java to Kava is a behavioural reflection add-on implemented as a demonstration of the capabilities of JavaAssist, a byte code rewriting tool based on structural reflection. Like Kava, this add-on adds hooks to the classes using byte code rewriting and has a similar meta level architecture of a binding between an object and a metaobject. However, it does not provide reflection on static members, on method invocation, or exception raising. Also it doesn't support

reflection on methods that have been inherited from a superclass, nor does it support the concept of a binding specification.

7. Conclusions and Future Work

Kava focuses on the behavioural changes programmers want to impose on third-party code instead of the messy structural changes that byte code transformation tools deal with. Kava allows adaptations to be developed, compiled and tested independently of the target code, then declaratively combined with the target code. This reduces the chance of error and makes that task of adapting the behaviour of third-party code more tractable.

Kava implements behavioural reflection in Java using byte code transformation as the underlying technique. This approach has allowed the creation of a tool unlike other reflective Java implementations is portable and can bring reflect on a wide range of runtime behaviours.

Kava is available for download from http://www.cs.ncl.ac.uk/research/dependability/ reflection. We are currently in the process of tuning the implementation to support lazy reification of context objects. We are also investigating the application of Kava to a case study based on flexible security for an enterprise modelling system.

Acknowledgements

This work has been supported by the UK Defence Evaluation Research Agency, grant number CSM/547/UA and also the ESPIRIT LTR project MAFTIA.

References

[1] U. Holzle, “Integrating Independently-Developed Components in Object-Oriented Languages,” ECOOP'93, Kaiserslautern, Germany, 1993.

[2] G. Czaijkowski and T. v. Eicken, “JRes : A Resource Accounting Interface for Java”, OOPSLA'98, 1998.

[3] R. Pandey and B. Hashii, “Providing Fine-Grained Access Control for Java Programs,” ECOOP'99, Lisbon, Portugal, 1999.

[4] R. Keller and U. Holzle, “Binary Component Adaptation,” ECOOP'98, 1998.

[5] G. A. Cohen and J. S. Chase, “Automatic Program Transformation with JOIE,” USENIX Annual Technical Symposium, New Orleans, Louisiana, 1998.

[6] M. Dahm, “Byte Code Engineering with the JavaClass API,” Friei Universitat, Berlin, Technical Report B-17-98, 1998.

[7] S. Chiba, “Load-time Structural Reflection in Java,” European Conference on Object-Oriented Programming, 2000.

[8] I. Welch and R. J. Stroud, “Kava - A Reflective Java Based on Bytecode Rewriting,” in *Reflection and Software Engineering*, vol. 1826, W. Cazzola, R. J. Stroud, and F. Tisato, Eds. Heidelberg, Germany: Springer-Verlag, 2000, pp. 157-169.

[9] P. Maes, “Concepts and Experiments in Computational Reflection,” OOPSLA'87, Orlando, Florida, 1987.

[10] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*: Massachusetts Institute of Technology, 1991.

[11] S. Matsuoka, T. Watanabe, and A. Yonezawa, “Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming,” ECOOP'91, 1991.

[12] R. J. Stroud and Z. Wu, “Using Metaobject Protocols to Implement Atomic Data Types,” ECOOP'95, Aarhus, Denmark, 1995.

[13] J.-C. Fabre, V. Nicomette, T. Perennou, Z. Wu, and R. J. Stroud, “Implementing Fault-tolerant Applications using Reflective Object-Oriented Programming,” FTCS-25, Pasadena, USA, 1996.

[14] M. Benantar, B. Blakley, and A. J. Nadain, “Approach to Object Security in Distributed SOM,” *IBM Systems Journal*, vol. 35, 1996.

[15] Sun Microsystems Inc., “Java Development Kit 1.3.0 Documentation,” , 2000.

[16] C. Zimmerman, “Metalevels, MOPs and What all the Fuzz is All About,” in *Advances in Object-Oriented Metalevel Architectures and Reflection*, C. Zimmermann, Ed.: CRC Press, 1996.

[17] Z. Wu and S. Schwiderski, “Reflective Java : The Design, Implementation and Applications,” , 1996.

[18] M. Tatsubori and S. Chiba, “Programming Support of Design Patterns with Compile-time Reflection,” Workshop on Reflective Programming in C++ and Java, 1998.

[19] IBM, “Bean Extender Documentation, version 2.0”, 1997.

[20] I. S. Welch and R. J. Stroud, “From Dalang to Kava - the Evolution of a Reflective Java Extension,” Second International Conference on Meta-Level Architectures and Reflection, Saint-Malo, France, 1999.

[21] M. Golm, “Design and Implementation of a Meta Architecture for Java”, M.Sc. Erlangen, 1997.

[22] J. de. O. Guimarães, “Reflection for Statically Typed Languages” ECOOP'98, 1998.

[23] A. Oliva, and L. E. Bizato, “The Design and Implementation of Guaraná,”, COOTS'99, 1999

[24] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura, “OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java,” ECOOP'2000, 2000.