Load-Time Structural Reflection in Java

Shigeru Chiba

Institute of Information Science and Electronics
University of Tsukuba
and Japan Science and Technology Corp.
chiba@is.tsukuba.ac.jp, chiba@acm.org

Abstract. The standard reflection API of Java provides the ability to introspect a program but not to alter program behavior. This paper presents an extension to the reflection API for addressing this limitation. Unlike other extensions enabling behavioral reflection, our extension called *Javassist* enables structural reflection in Java. For using a standard Java virtual machine (JVM) and avoiding a performance problem, Javassist allows structural reflection only before a class is loaded into the JVM. However, Javassist still covers various applications including a language extension emulating behavioral reflection. This paper also presents the design principles of Javassist, which distinguish Javassist from related work.

1 Introduction

Java is a programming language supporting reflection. The reflective ability of Java is called the reflection API. However, it is almost restricted to introspection, which is the ability to introspect data structures used in a program such as a class. The Java's ability to alter program behavior is very limited; it only allows a program to instantiate a class, to get/set a field value, and to invoke a method through the API.

To address the limitations of the Java reflection API, several extensions have been proposed. Most of these extensions enable behavioral reflection, which is the ability to intercept an operation such as method invocation and alter the behavior of that operation. If an operation is intercepted, the runtime systems of those extensions call a method on a *metaobject* for notifying it of that event. The programmer can define their own version of metaobject so that the metaobject executes the intercepted operation with customized semantics, which implement a language extension for a specific application domain such as fault tolerance [9].

However, behavioral reflection only provides the ability to alter the behavior of operations in a program but not provides the ability to alter data structures used in the program, which are statically fixed at compile time (or, in languages like Lisp, when they are first defined). The latter ability called structural reflection allows a program to change, for example, the definition of a class, a function, and a record on demand. Some kinds of language extensions require this ability for implementation and thus they cannot be implemented with a straightforward program using behavioral reflection; complex programming tricks are often needed.

To simply implement these language extensions, this paper presents *Javassist*, which is a class library for enabling structural reflection in Java. Since portability is important in Java, we designed a new architecture for structural reflection, which can be implemented without modifying an existing runtime system or compiler. Javassist is a Java implementation of that architecture. An essential idea of this architecture is that structural reflection is performed by bytecode transformation at compile-time or load time. Javassist does not allow structural reflection after a compiled program is loaded into the JVM. Another feature of our architecture is that it provides source-level abstraction: the users of Javassist do not have to have a deep understanding of the Java bytecode. Our architecture can also execute structural reflection faster than the compile-time metaobject protocol used by OpenC++ [3] and OpenJava [20].

In the rest of this paper, we first overview previous extensions enabling behavioral reflection in Java and point out limitations of those extensions. Then we present the design of Javassist in Section 3 and show typical applications of Javassist in Section 4. In Section 5, we compare our architecture with related work. Section 6 is conclusion.

2 Extensions to the Reflection Ability of Java

The Java reflection API dose not provide the full reflective capability. It does not enable alteration of program behavior but it only supports introspection, which is the ability to introspect data structures, for example, inspecting a class definition. This design decision was acceptable because implementing the full capability was difficult without a decline in runtime performance. An implementation technique using partial evaluation has been proposed [17,2] but the feasibility of this technique in Java has not been clear.

However, several extensions to the Java reflection API have been proposed. To avoid performance degradation, most of these extensions enable restricted behavioral reflection. They only allow alteration of the behavior of specific kinds of operations such as method calls, field accesses, and object creation. The programmers can select some of those operations and alter their behavior. The compilers or the runtime systems of those extensions insert *hooks* in programs so that the execution of the selected operations is intercepted. If these operations are intercepted, the runtime system calls a method on an object (called *a metaobject*) associated with the operations or the target objects. The execution of the intercepted operation is implemented by that method. The programmers can define their own version of metaobject for implementing new behavior of the intercepted operations.

The runtime overheads due to this restricted behavioral reflection are low since only the execution of the intercepted operations involves a performance penalty and the rest of the program runs without any overheads. Especially, if hooks for the interception are statically inserted in a program during compilation, the runtime overheads are even lowered. To statically insert hooks, Reflective Java [22] performs source-to-source translation before compilation and Kava [21] performs bytecode-level transformation when a program is loaded into the JVM. MetaXa [16,11] internally performs bytecode-level transformation with a customized JVM. It uses a customized just-in-time compiler (JIT) for improving the execution speed of the inserted hooks. This hook-insertion technique is well known and has been applied to other languages such as C++ [4].

Although the restricted behavioral reflection is useful for implementing various language extensions, there are some kinds of extensions that cannot be intuitively implemented with that kind of reflection. An example of these extensions is binary code adaptation (BCA) [13], which is a mechanism for altering a class definition in binary form to conform changes of the definitions of other classes. Suppose that we write a program using a class library obtained from a third party. For example, our class Calendar implements an interface Writable included in that class library:

```
class Calendar implements Writable {
  public void write(PrintStream s) { ... }
}
```

The class Calendar implements method `write()` declared in the interface Writable.

Then, suppose that the third party gives us a new version of their class library, in which the interface Writable is renamed into Printable and it declares a new method `print()`. To make our program conform this new class library, we must edit the definitions of all our classes implementing Writable, including Calendar:

```
class Calendar implements Printable {
  public void write(PrintStream s) { ... }
  public void print() { write(System.out); }
}
```

The interface of Calendar is changed into Printable and method `print()` is added.

BCA automates this adaptation; it automatically alters class definitions in binary form according to a configuration file specifying how to alter them. Note that the method body of `print()` is identical among all the updated classes since `print()` can be implemented with the functionality already provided by `write()` for the old version. If that configuration file is supplied by the library developer, we can run our program without concern about evolution of the class library.

Unfortunately, implementing BCA with behavioral reflection is not intuitive or straightforward. Since behavioral reflection cannot directly provide the ability to alter data structures such as a class definition or construct a new data structure, these reflective computation must be indirectly implemented. For example, the implementation of BCA with behavioral reflection defines a metaobject indirectly performing the adaptation specified by a given configuration file. For the above example, this metaobject is made to be associated with Calendar and it watches method calls on Calendar objects. If the method `print()` is called, the metaobject intercepts that method call and executes the computation corresponding to `print()` instead of the Calendar object. The metaobject also intercepts runtime type checking so that the JVM recognizes Calendar as a subtype of Printable. Recall that Java is a statically typed language and the original Calendar is a subtype of Writable.

The ability to alter data structures used in a program is called structural reflection, which has not been directly supported by previous systems. Although a number of language extensions are more easily implemented with structural reflection than with behavioral reflection, the previous systems have not been addressing those extensions. They have been too much focused on language extensions that can be implemented by altering the behavior of method calls and so on.

3 Javassist

To simply implement language extensions like BCA shown in the previous section, we developed Javassist, which is our extension to the Java reflection API and enables structural reflection instead of behavioral one. Javassist is based on our new architecture for structural reflection, which can be implemented without modifying an existing runtime system or a compiler.

3.1 Implementations of Structural Reflection

Structural reflection is the ability to allow a program to alter the definitions of data structures such as classes and methods. It has been provided by several languages such as Smalltalk [10], ObjVlisp [6], and CLOS [14]. These languages implement structural reflection with support mechanisms embedded in runtime systems. Since the runtime systems contain internal data representing the definitions of data structures such as a class, the support mechanisms allow a program to directly read and change those internal data and thereby execute structural reflection on the correspondent data structures.

We could not accept this implementation technique for Javassist since it needs to modify a standard JVM but portability is important in Java. Furthermore, a naive application of this technique to Java would cause serious performance degradation of the JVM because this technique makes it difficult for runtime systems to employ optimization techniques based on static information of executed programs. Since a program may be altered at runtime, efficient dynamic recompilation is required for redoing optimization on demand. For example, method inlining is difficult to perform. If an inlined method is altered at runtime with structural reflection, all the inlined code must be updated. To do this, the runtime system must record where the code is inlined. This will spend a large amount of memory space. Another example is the "v-table" technique used for typical C++ implementations [8]. This technique statically constructs method dispatch tables so that invoked methods are quickly selected with a constant offset in the tables. If a new method is added to a class at runtime, then the dispatch tables may be updated and all offsets in the tables may be recomputed. Since the dynamic recompilation technique has been used so far for gradually optimizing "hot spots" of compiled code at runtime [12], it has been assuming that a program is never changed at runtime. Effectiveness of dynamic recompilation without this assumption is an open question.

Another problem is correctness of types. Since Java is a statically typed language, a variable of type X must be bound to an object of X or a subclass Y of X. If a program can freely access and change the internal data of the JVM, it may dynamically change the super class of Y from X to another class. This change causes a type error for the binding between a variable of type X and an object of Y. To address this problem, extra runtime type checks or restrictions on the range of structural reflection are needed.

3.2 Load-Time Structural Reflection

To avoid the problems mentioned above, we designed a new architecture for structural reflection; it does not need to modify an existing runtime system or a compiler. On the other hand, it enables structural reflection only before a program is loaded into a runtime system, that is, at load time. Javassist is a class library enabling structural reflection based on this architecture. In Java, the bytecode obtained by compilation of a program is stored in *class files*, each of which corresponds to a distinct class. Javassist performs structural reflection by translating alterations by structural reflection into equivalent bytecode transformation of the class files. After the transformation, the modified class files are loaded into the JVM and then no alterations are allowed after that. Thereby, Javassist can be used with a standard JVM, which may use various optimization techniques.

Javassist is used with a user class loader. Java allows programs to define their own versions of class loader, which fetch a class file from a not-standard resource such as a network. A typical definition of the class loader is as follows:

```
class MyLoader extends ClassLoader {
  public Class loadClass(String name) {
    byte[] bytecode = readClassFile(name);
    return resolveClass(defineClass(bytecode));
  }

  private byte[] readClassFile(String name) {
    // read a class file from a resource.
  }
}
```

The methods `defineClass()` and `resolveClass()` are inherited from ClassLoader. They request the JVM to load a class constructed from the bytecode given as an array of `byte`. The returned value is a Class object representing the loaded class. Once a class X is manually loaded with an instance of MyLoader, all classes referenced by that class X are loaded through that class loader. The

JVM automatically calls `loadClass()` on that class loader for loading them on demand.

Javassist helps `readClassFile()` shown above obtain the bytecode of a requested class. It can be regarded as a class library for reading bytecode from a class file and altering it. However, unlike similar class libraries such as the JavaClass API [7] and JOIE [5], Javassist provides source-level abstraction so that it can be used without knowledge of bytecode or the data format of the class file. Also, Javassist was designed to make it difficult to wrongly produce a class file rejected by the bytecode verifier of the JVM.

3.3 The Javassist API

We below present the overview of the Javassist API.

Reification and Reflection: The first step of the use of Javassist is to create a CtClass (compile-time class) object representing the bytecode of a class loaded into the JVM. This step is for reifying the class to make it accessible from a program. If `stream` is an InputStream for reading a class file (from a local disk, memory, a network, etc.), then:

```
CtClass c = new CtClass(stream);
```

creates a new CtClass object representing the bytecode of the class read from the class file, which contains enough symbolic information to reify the class. Also, the constructor of CtClass can receive a String class name instead of an InputStream. If a String class name is given, Javassist searches a class path and finds an InputStream for reading a class file.

One can call various methods on the CtClass object for introspecting and altering the class definition. Changes of the class definition are reflected on the bytecode represented by that object. To obtain the bytecode for loading the altered class into the JVM, method `toBytecode()` is called on that object:

```
byte[] bytecode = c.toBytecode();
```

Loading the obtained bytecode into the JVM is regarded as the step for reflecting the CtClass object on the base level. Javassist provides several other methods for this step. For example, method `compile()` writes bytecode to a given output stream such as a local file and a network. Method `load()` directly loads the class into the JVM with a class loader provided by Javassist. It returns a Class object representing the loaded class. Recall that Class is included in the Java reflection API while CtClass is in Javassist.

Note that Javassist does not provide any framework for specifying how and what classes are processed with Javassist. The programmer of the class loader has freedom with respect to this framework. For example, the class loader may process classes with Javassist only if they are specified by a configuration file read at the beginning. It may process them according to a *hard-coded* algorithm.

Table 1. Methods in CtClass for introspection

Method		Description
`String`	`getName()`	gets the class name
`int`	`getModifiers()`	gets the class modifiers such as `public`
`boolean`	`isInterface()`	determines whether this object represents a class or an interface
`CtClass`	`getSuperclass()`	gets the super class
`CtClass[]`	`getInterfaces()`	gets the interfaces
`CtField[]`	`getDeclaredFields()`	gets the fields declared in the class
`CtMethod[]`	`getDeclaredConstructors()`	gets the constructors declared in the class
`CtMethod[]`	`getDeclaredMethods()`	gets the methods declared in the class

Javassist allows a user class loader to define a new class from scratch without reading any class file. This is useful if a program needs to dynamically define a new class on demand. To do this, a CtClass object must be created as follows:

```
CtClass c2 = new CtNewClass();
```

The created object `c2` represents an empty class that has no methods or fields although methods and fields can be added to the class later through the Javassist API shown below. If `toBytecode()` is called on this object, then it returns the bytecode corresponding to that empty class.

Introspection: Javassist provides several methods for introspecting the class represented by a CtClass object. This part of the Javassist API is compatible with the Java reflection API except that Javassist does not provide methods for creating an instance or invoking a method because these methods are meaningless at load time. Table 1 lists selected methods for introspection.

CtClass objects returned by `getSuperclass()` and `getInterfaces()` are constructed from class files found on a class path. They represent the original class definitions and thus accept only introspection but not alteration. To alter a class, another CtClass object must be explicitly created with the `new` operator. Modifications to this object have no effect on the CtClass object returned by `getSuperclass()` or `getInterfaces()`. For example, suppose that a class C inherits from a class S. If a CtClass object for S is created with `new` and a method `m()` is added to that object, this modification is not reflected on the object returned by `getSuperclass()` on a CtClass object for C. The class C inherits `m()` from S only if the CtClass object created with `new` is converted into bytecode and loaded into the JVM.

The information about fields and methods is provided by objects separate from the CtClass object; it is provided by CtField objects obtained by `getDeclaredFields()` and CtMethod objects obtained by `getDeclaredMethods()`, respectively. The information about a constructor is also provided by a CtMethod object. Table 2 lists methods in CtField and CtMethod for introspection.

Table 2. Methods in CtField and CtMethod for introspection

Method	in CtField	Description
`String`	`getName()`	gets the field name
`CtClass`	`getDeclaringClass()`	get the class declaring the field
`int`	`getModifiers()`	gets the field modifiers such as `public`
`CtClass`	`getType()`	get the field type
Method	**in CtMethod**	**Description**
`String`	`getName()`	gets the method name
`CtClass`	`getDeclaringClass()`	get the class declaring the method
`int`	`getModifiers()`	gets the method modifiers such as `public`
`CtClass[]`	`getParameterTypes()`	gets the types of the parameters
`CtClass[]`	`getExceptionTypes()`	gets the types of the exceptions that the method may throw
`boolean`	`isConstructor()`	returns `true` if the method is a constructor
`boolean`	`isClassInitializer()`	returns `true` if the method is a class initializer

Table 3. Methods for alteration

Method	in CtClass	Description
`void`	`bePublic()`	make the class `public`
`void`	`beAbstract()`	make the class `abstract`
`void`	`notFinal()`	remove the `final` modifier from the class
`void`	`setName(String name)`	change the class name
`void`	`setSuperclass(CtClass c)`	change the super class
`void`	`setInterfaces(CtClass[] i)`	change the interfaces
`void`	`addConstructor(...)`	add a new constructor
`void`	`addDefaultConstructor()`	add the default constructor
`void`	`addAbstractMethod(...)`	add a new `abstract` method
`void`	`addMethod(...)`	add a new method
`void`	`addWrapper(...)`	add a new wrapped method
`void`	`addField(...)`	add a new field
Method	**in CtField**	**Description**
`void`	`bePublic()`	make the field `public`
Method	**in CtMethod**	**Description**
`void`	`bePublic()`	make the method `public`
`void`	`instrument(...)`	modify a method body
`void`	`setBody(...)`	substitute a method body
`void`	`setWrapper(...)`	substitute a method body

Alteration: A difference between Javassist and the standard Java reflection API is that Javassist provides methods for altering class definitions. Several methods for alteration are defined in CtClass (Table 3). These methods are categorized into methods for changing class modifiers, methods for changing class hierarchy, and methods for adding a new member. They were carefully selected to satisfy our design goals.

Our design goals are three. (1) The first goal is to provide source-level abstraction for programmers. Javassist was designed so that programmers can use it without knowledge of the Java bytecode. (2) The second goal is to execute structural reflection as efficiently as possible. (3) The last goal is to help programs perform structural reflection in a safe manner in terms of types.

As for the first goal, the most significant design decision was how programmers specify a method body. Suppose that a new method is added to a class. If a sequence of bytecode is used for specifying the body of that method, the programmers would get great flexibility but have to learn details of bytecode. To achieve the first goal, Javassist allows to copy a method body from another existing method although this design decision restricts the flexibility of the added method. The copied bytecode sequence is adjusted to fit the destination method. For example, the bytecode for accessing a member through the `this` variable contains a symbolic reference to the type of `this`. This reference is replaced with one to the class declaring the destination method.

Despite the well-known quasi-equivalence between Java source code and bytecode, the correspondence between source-level and bytecode-level alterations are not straightforward. Hiding the gap between the two levels from programmers is also a part of the first goal.

For example, `setName()` renames a class but it also substitutes the new name for all occurrences of the old name in the definition of that class, including method signatures and bodies. Modifying a single constant-pool item never performs this substitution. If a constructor calls another constructor in the same class (if it executes `this()`), then the bytecode of the former constructor is modified since the bytecode contains a symbolic reference to the name of the class declaring the latter constructor. This reference must be modified to indicate the new name.

`setSuperclass()` performs similar substitution. If it is called, all occurrences of the old super class name is replaced with a new name and all constructors are modified so that they call a constructor in the new super class. However, there is an exception to this substitution. If the name of the original super class is java.lang.Object (the root of the class hierarchy), `setSuperclass()` does not perform the substitution except it modifies constructors. This is because java.lang.Object is often used for representing any class. For example, although `addElement()` in java.util.Vector takes a parameter of class java.lang.Object, which is the super class of java.util.Vector, this never means that `addElement()` takes an instance of the super class.

The second design goal is to reduce overheads due to class loading with Javassist. Since we will use Javassist for implementing a mobile-agent system, in which Javassist inserts security-check code into bytecode, Javassist must transform bytecode received through a network as efficiently as possible. Mobile agents frequently move among hosts and thus we cannot ignore the loading time of the bytecode implementing the mobile agents.

Our design decision on how programmers specify a method body was influenced by the second goal as well as the first one. Javassist does not use source code for specifying the body of an added method. If source code is used, it must be compiled *on the fly* when a class is loaded into the JVM. A naive implementation of this source-code approach would produce a complete class definition including the added method at source level and then compile it with a Java compiler such as `javac`. As we show later, however, this implementation implies serious performance penalties. To achieve practical efficiency, we need a special compiler that can quickly compile only a method body. We did not adopt the source-code approach because of limitations of our resources. Instead, Javassist allows to copy a pre-compiled method body from a class to another. This approach does not imply overheads due to source-code compilation at load time.

The third design goal is to prevent programs to wrongly produce a class including type incorrectness. To achieve this goal, Javassist allows only limited kinds of alteration of class definitions. In general, reflective systems should impose some restrictions on structural reflection so that programs do not falsely collapse themselves with reflection. Suppose that a reflective system allows to remove a field from a class at runtime. If there are already instances of that class, is it appropriate that the system simply discards the value of the removed field of those instances?

Since erroneous bytecode produced with Javassist is rejected by the bytecode verifier, it can never damage the JVM. However, restricting the reflective capability of Javassist is still necessary because it is often awkward to correct a program producing erroneous bytecode. For this reason, Javassist does not provide methods for removing a method or a field from a class because they cause type incorrectness if there is a method accessing the removed method or field. Javassist also imposes restrictions on the class passed to `setSuperclass()`, which is a method for changing a super class. The new super class must be a subclass of the original super class since there may be methods that implicitly cast an instance of that class to the original super class. Of course, the new super class must not be `final`. Furthermore, Javassist does not provide a method for changing the parameters of a method. Programmers are recommended to add a new method with the same name but with different parameters.

Adding a new member: Javassist provides methods for adding a new method to a class. To avoid the abstraction and performance problems mentioned above, `addMethod()` receives a `CtMethod` object, which specifies a method body. The signature of `addMethod()` is as shown below:

```
void addMethod(CtMethod m, String name, ClassMap map)
```

`name` specifies the name of the added method. The method body is copied from a given method `m`. Since a method body is copied from an existing compiled method, no source-code compilation is needed at load time or no raw bytecode is given to `addMethod()`. Programmers can describe a method body in Java and compile it in advance. Javassist reads the bytecode of the compiled method and adds it to another class. This improves execution performance of Javassist since a compiler is not run at load time.

When a method body is copied, some class names appearing in the body can be replaced according to a hash table `map`.[1] For example, programmers can declare a class XVector:

```
public class XVector extends java.util.Vector {
  public void add(X e) {
    super.addElement(e);
  }
}
```

and copy the method `add()` into a class StringVector:

```
CtMethod m = /* method add() in XVector */;
CtClass c = /* class StringVector */;
ClassMap map = new ClassMap();
map.put("X", "java.lang.String");
c.addMethod(m, "addString", map);
```

The class name java.lang.String is substituted for all occurrences of the class name X in `add()`. The added method is as follows:

```
public void addString(java.lang.String e) {
  super.addElement(e);
}
```

Javassist provides another method `addWrapper()` for adding a new method. It allows more generic description of a method body:

```
void addWrapper(int modifiers, CtClass returnType, String name,
                CtClass[] parameters, CtClass[] exceptions,
                CtMethod body, ConstParameter constParam)
```

The first five parameters specify the modifiers, the return type, the method name, the parameter types, and the exceptions that the method may throw. The body of the added method is copied from the method specified by `body`. No matter what the signature of the added method is, the method specified by `body` must have the following signature:

```
Object m(Object[] args, value-type constValue)
```

To fill the gap between this signature and the signature of the added method, `addWrapper()` implicitly wraps the copied method body in *glue* code, which constructs an array of actual parameters passed to the added method and assigns it to `args` before executing the copied method body. The glue code also sets `constValue` to a constant value specified by `constParam` passed to `addWrapper()`. In the current version of Javassist, an integer value or a `String`

[1] At least, `addMethod()` replaces all occurrences of the name of the class declaring the copied method. Even if that class name does not appear at source level, the corresponding bytecode may include references to it.

object can be specified for the constant value. For example, this constant value can be used to pass the name of the added method.

The value returned by the copied method body is an Object object. The glue code also converts it into a value of the type specified by `returnType`. Then it returns the converted value to the caller to the added method. If type conversion fails, then an exception is thrown. Although methods added by `addWrapper()` involve runtime overheads due to type conversion, a single method body can be used as a template of multiple methods receiving a different number of parameters. Examples of the use of `addWrapper()` are shown in Section 4.

Javassist also provides a method for adding a new field to a class:

```
void addField(int modifiers, CtClass type, String fieldname,
              String accessor, FieldInitializer init)
```

If `accessor` is not `null`, this method also adds an accessor method, which returns the value of the added field. The name of the accessor is specified by `accessor`. Moreover, the last parameter `init` specifies the initial value of the added field. The initial value is either one of parameters passed to a constructor, a newly created object, or the result of a call to a static method.

Altering a method body: Although Javassist does not allow to remove a method from a class, it provides methods for changing a method body. `setBody()` and `setWrapper()` in CtMethod substitute a given method body for an original body:

```
void setBody(CtMethod m, ClassMap map)
void setWrapper(CtMethod m, ConstParameter param)
```

They correspond to `addMethod()` and `addWrapper()` respectively. `setBody()` copies a method body from a given method `m`. Some class names appearing in the body are replaced with different names according to `map`. `setWrapper()` also copies a method body from `m` but it wraps the copied body in glue code. The signature of `m` must be:

`Object m(Object[] args,` *value-type* `constValue)`

Javassist also provides a method for modifying expressions in a method body. `instrument()` in CtMethod performs this modification:

```
void instrument(CodeConverter converter)
```

The parameter `converter` specifies how to instrument a method body. The CodeConverter object can perform various kinds of instrumentation. Table 4 lists methods provided by the current implementation of Javassist. They direct a CodeConverter object to replace a specific kind of expressions with *hooks*, which invoke static methods for executing the expressions in a customized manner. The idea of CodeConverter came from C++'s operator overloading. CodeConverter was

Table 4. Methods in CodeConverter

Method	Description
`void redirectFieldAccess()`	change a field-access expression to access a different field.
`void replaceNew()`	replace a `new` expression with a `static` method call.
`void replaceFieldRead()`	replace a field-read expression with a `static` method call.
`void replaceFieldWrite()`	replace a field-write expression with a `static` method call.

designed for safely altering the behavior of operators such as `new` and `.` (dot) independently of the context.

For example, expressions for instantiating a specific class can be replaced with expressions for calling a static method. Suppose that variables `xclass` and `yclass` represent class X and Y, respectively. Then a program:

```
CtMethod m = ... ;
CodeConverter conv = new CodeConverter();
conv.replaceNew(xclass, yclass, "create");
m.instrument(conv);
```

instruments the body of the method represented by the CtMethod object `m`. All expressions for instantiating the class X such as:

```
new X(3, 4);
```

are translated into expressions for calling a static method `create()` declared in the class Y:

```
Y.create(3, 4);
```

The parameters to the `new` expression are passed to the static method.

Reflective class loader: The class loader provided by Javassist allows a loaded program to control the class loading by that class loader. If a program is loaded by Javassist's class loader L and it includes a class C, then it can intercept the loading of C by L to self-reflectively modify the bytecode of C (Figure 1). For avoiding infinite recursion, while the loading of a class is intercepted, further interception is prohibited. The `load()` method in CtClass requires that a program is loaded by Javassist's class loader although the other methods work without Javassist's class loader.

Java's standard class loader never allows this self-reflective class loading for security reasons. If it is allowed, a program may change some `private` fields to `public` ones at load time for reading hidden values. Furthermore, in Java, if a program creates a class loader and loads a class C with that class loader, the loaded class is regarded as a different one from the class denoted by the name C appearing in that program. The latter class is loaded by the class loader that loaded the program.

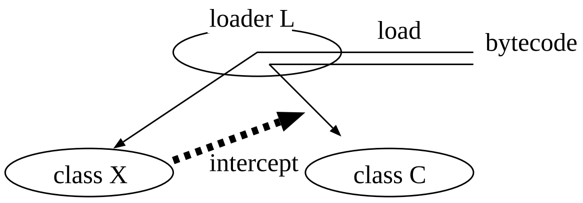

Fig. 1. Javassist's class loader allows self-reflective class loading

Using Javassist without a class loader: Javassist can be used without a user class loader. There are three kinds of usage of Javassist: with a user class loader, with a web server, and off line.

For security reasons, an applet is usually prohibited from using a user class loader. However, we can write an applet working with Javassist if we use a web server as a replacement of a user class loader. Since classes used in an applet are loaded from a web server into the JVM of a web browser, we can customize the web server so that it runs Javassist for processing the classes before sending them to the web browser. Javassist includes a simple web server written in Java as a basis for such customization. We can extend it to perform structural reflection with Javassist. The program of the customized web server would be as follows:

```
for (;;) {
  receive an http request from a web browser.
  CtClass c = new CtClass(the requested class);
  do structural reflection on c if needed.
  byte[] bytecode = c.toBytecode();
  send the bytecode to the web browser.
}
```

Before sending a requested class to a web browser, it performs structural reflection on the class according to the algorithm, for example, given as a configuration file.

Another usage of Javassist is "off line". We can perform structural reflection on a class and overwrite the original class file of that class with the bytecode obtained as the result. The altered class can be later loaded into the JVM without a user class loader. The following is an example of the off-line use of Javassist:

```
CtClass c = new CtClass("Rectangle");
do structural reflection on c if needed.
c.compile();     // writes bytecode on the original class file.
```

This program performs structural reflection on class Rectangle and overwrites the class file of that class with the bytecode obtained by `c.toBytecode()`.

4 Examples

This section shows three applications of Javassist. We illustrate that Javassist can be used to implement non-trivial alteration required by these applications despite the level of the abstraction.

4.1 Binary Code Adaptation

The mechanism of binary code adaptation (BCA) [13] automatically alters class definitions according to a file written by the users, called a delta file:

```
delta class implements Writable {
  rename Writable Printable;
  add public void print() { write(System.out); }
}
```

This delta file specifies adaptation that we mentioned in Section 2.

If Javassist is used, the implementor of BCA has only to write a parser of delta file and a user class loader performing adaptation with Javassist. For example, the parser translates the delta file shown above into the Java program shown below:

```
class Exemplar implements Printable {
  public void write(PrintStream s) { /* dummy */ }
  public void print() { write(System.out); }
}

class Adaptor {
  public void adapt(CtClass c) {
    CtMethod printM = /* method print() in Exemplar */;
    CtClass[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
      if (interfaces[i].getName().equals("Writable")) {
        interfaces[i] = CtClass.forName("Printable");
        c.setInterfaces(interfaces);
        c.addMethod(printM, new ClassMap());
        return;
      }
  }
}
```

The class Exemplar is compiled together with Adapter in advance so that `adapt()` can obtain a `CtMethod` object representing `print()`. `adapt()` uses the reification and introspection API of Javassist for obtaining it. It first constructs a `CtClass` object representing Exemplar and then obtains the `CtMethod` object by `getDeclaredMethods()` in CtClass. The class file for Exemplar is automatically found by Javassist on the class path used for loading Adapter.

The user class loader calls `adapt()` in Adaptor whenever a class is loaded into the JVM. It creates a CtClass object representing the loaded class and calls `adapt()` with that object. The method `adapt()` performs adaptation if the loaded class implements Writable. Then the user class loader converts the CtClass object into bytecode and loads into the JVM.

Note that this implementation is more intuitive than the implementation with behavioral reflection. Moreover, it is simpler than the implementation without reflection since the implementor does not have to care about low-level bytecode transformation. If the users of BCA can directly write the classes Exemplar and Adaptor instead of a delta file, then the implementation would be much simpler since we do not need the parser of delta file.

4.2 Behavioral Reflection

Behavioral reflection enabled by MetaXa [16,11] and Kava [21] can be implemented with an approximately 750-line program (including comments) using Javassist. A key idea of their implementations is to insert *hooks* in a program when a class is loaded into the JVM. We below see an overview of a user class loader performing this insertion with Javassist.

Let a metaobject be an instance of MyMetaobject, which is a subclass of Metaobject:

```
public class MyMetaobject extends Metaobject {
  public Object trapMethodcall(String methodName, Object[] args) {
    /* called if a method call is intercepted. */ }
  public Object trapFieldRead(String fieldName) {
    /* called if the value of a field is read. */ }
  public void trapFieldWrite(String fieldName, Object value) {
    /* called if a field is set. */ }
}
```

If field accesses and method calls on an instance of C:

```
public class C {
  public int m(int x) { return x + f; }
  public int f;
}
```

are intercepted by the metaobject, then the user class loader alters the definition of the class C into the following:[2]

```
public class C implements Metalevel {
  public int m(int x) { /* notify a metaobject */ }
  public int f;
  private Metaobject _metaobject = new MyMetaobject(this);
  public Metaobject _getMetaobject() { return _metaobject; }
  public int orig_m(int x) { return x + f; }
  public static int read_f(Object target) {
      /* notify a metaobject */ }
  public static void write_f(Object target, int value) {
      /* notify a metaobject */ }
}
```

where the interface Metalevel declares the method _getMetaobject().

[2] For simplicity, this implementation ignores static members although extending the implementation for handling static members is possible within the ability of Javassist.

```
class Exemplar {
  private Metaobject _metaobject;

  public Object trap(Object[] args, String methodName) {
    return _metaobject.trapMethodcall(methodName, args);
  }

  public static Object trapRead(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    return target._getMetaobject().trapFieldRead(name);
  }

  public static Object trapWrite(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    Object value = args[1];
    target._getMetaobject().trapFieldWrite(name, value);
  }
}
```

Fig. 2. Class Exemplar

This alteration can be performed within the ability of Javassist. The interface Metalevel is added by `setInterfaces()` in CtClass. The field `_metaobject` and the accessor `_getMetaobject()` are added by `addField()` in CtClass.

For intercepting method calls, the user class loader first makes a copy of every method in C by calling `addMethod()` in CtClass. For example, it adds `orig_m()`[3] as a copy of `m()`. Then it replaces the body of every method in C with a copy of the body of the method `trap()` in Exemplar (see Figure 2). This modification is performed by `setWrapper()` in CtMethod. The gap between the signatures of `m()` and `trap()` is filled by `setWrapper()`. The substituted method body notifies a metaobject of interception. The first parameter `args` is a list of actual parameters and the second one `name` is the name of the copy of the original method such as `"orig_m"`. These two parameters are used for the metaobject to invoke the original method through the Java reflection API.

For intercepting field accesses, the user class loader instruments the bodies of methods in all classes. All accesses to a field `f` in C are translated into calls to a `static` method `read_f()` or `write_f()`. This instrumentation is performed by `instrument()` in CtMethod and `replaceFieldRead()` and `replaceField-Write()` in CodeConverter. The methods `read_f()` and `write_f()` notify a metaobject of the accesses. They are added by `addWrapper()` in CtClass as copies of `trapRead()` and `trapWrite()` in Exemplar. The gap between the signatures of `read_f()` (or `write_f()`) and `trapRead()` (or `trapWrite()`) is filled by `addWrapper()`. For example, actual parameters to `read_f()` are converted into the first parameter `args` to `trapRead()`. The second parameter `name` to `trapRead()` is the name of the accessed field such as `"f"`.

[3] If a method name is overloaded, a copy of each method must be given a different name such as `orig_m1()`, `orig_m2()`, ...

4.3 Remote Method Invocation

Generating stub code for remote method invocation is another application of Javassist. A Java program cannot directly call a method on a remote object on a different computer. It needs the Java RMI tools generating stub code, which translates a method call into lower-level network data transfer such as TCP/IP communication. However, the Java RMI tools are compile-time ones; a program must be processed by the RMI compiler, which generates and saves stub code on a local disk. Also, a program using the Java RMI must be subject to a protocol (i.e. API) specified by the Java RMI.

Javassist allows programmers to develop their own version of the RMI tools, which specify a customized protocol and produce stub code at either compile-time or even runtime. Suppose that an applet needs to call a method on a Counter object on a web server written in Java. For remote method invocation, the applet needs stub code defining a proxy object of the Counter object, which has the same set of methods as the Counter object. If the Counter object has a method `setCount()`, the proxy object also has a method `setCount()` with the same signature. However, the method on the proxy object serializes given parameters and sends them to the web server, where `setCount()` is invoked on the Counter object with the received parameters.

This stub code can be generated at runtime with Javassist at the server side and it can be sent on demand to the applet side. The applet programmer can easily write the applet without concern about low-level network programming. The stub code for accessing the Counter object is as follows:

```
public class ProxyCounter {
  private RmiStream rmi;
  public ProxyCounter(int objectRef) {
    rmi = new RmiStream(objectRef);
  }
  public int setCount(int value) { /* remote method invocation */ }
}
```

An instance of ProxyCounter is a proxy object. An RmiStream object handles low-level network communication. The class RmiStream is provided by a runtime support library.

ProxyCounter can be defined within the confines of Javassist. The field `rmi` is added by `addField()` in CtClass and the initialization of `rmi` in a constructor can be specified by a FieldInitializer object passed to `addField()`.

The method `setCount()` is added by `addWrapper()` in CtClass as a copy of the method `invoke()` in Exemplar shown below:

```
class Exemplar {
  private RmiStream rmi;
  Object invoke(Object[] args, String methodName) {
    return rmi.rpc(methodName, args);
  }
}
```

The gap between the signatures of `setCount()` and `invoke()` is filled by `addWrapper()`. If `setCount()` is called, the actual parameter `value` is converted into an array of Object and assigned to `args`. `methodName` is set to a method name `"setCount"`[4]. Then `rpc()` is called on the RmiStream object for serializing the given parameters and sends them to the web server. Note that the parameters can be serialized within the ability of the standard Java if they are converted into an array of Object.

Stub code generation is another example, which is not straightforward to implement with behavioral reflection. In a typical implementation with behavioral reflection, a proxy object is an instance of the class Counter although all method calls on the proxy object are intercepted by a metaobject and forwarded to a remote object; the class ProxyCounter is not produced. Therefore, if the proxy object is created, a constructor declared in Counter is called and may cause fatal side-effects since the class Counter is defined as a class at the server side but the proxy object is not at that side.

5 Related Work

Reflection in Java: MetaXa [16,11] and Kava [21] enable behavioral reflection in Java whereas Javassist enables structural reflection. They are suitable for implementing different kinds of language extensions. However, Javassist indirectly covers applications of MetaXa and Kava since a class loader providing functionality equivalent to MetaXa and Kava can be implemented with Javassist as we showed in Section 4.2.

Although Kava performs bytecode transformation of class files before the JVM loads them as Javassist does, they only insert hooks for interception in bytecode but do not run metaobjects at that time. They enable reflection at runtime and their ability is not structural reflection but the restricted behavioral reflection.

The Java reflection API was recently extended in the JDK 1.3 beta to partially enable behavioral reflection [19]. The new API allows a program to dynamically define a proxy class implementing given interfaces. An instance of this proxy class delegates all method invocations to another object through a type-independent interface.

Javassist is not the first system enabling structural reflection in Java. For example, Kirby et al proposed a system enabling structural reflection (they called it linguistic reflection) in Java although their system only allows to dynamically define a new class but not to alter a given class definition at load time [15]. With their system, a Java program can produce a source file of a new class, compile it with an external compiler such as `javac`, and load the compiled class with a user class loader. They reported that their system could be used for defining a class optimized for a given runtime condition.

[4] If a method name is overloaded, this should be `setCount1`, `setCount2`, ... for distinction.

Compile-time metaobject protocol: The compile-time metaobject protocol [3] is another architecture enabling structural reflection without modifying an existing runtime system. OpenJava [20] is a Java implementation of this architecture. As Javassist does, it restricts structural reflection within the time before a class is loaded into the JVM although it was designed mainly for off-line use at compile time. However, OpenJava is source-code basis although Javassist is bytecode basis; OpenJava reads source code for creating an object representing a class, a method, or a field. Alteration to the object is translated into corresponding transformation of the source code. The bytecode for the altered class is obtained by compiling the modified source code. Since OpenJava is source-code basis, it can deal with syntax extensions within a framework of structural reflection. For example, one can extend the syntax of class declaration and make it possible to add an annotation to a class declaration.

On the other hand, the source-code basis means that OpenJava needs the source file of every processed class whereas Javassist needs only a class file (compiled binary). This is a disadvantage because source files are not always available if the class is provided by a third party. OpenJava also involves a performance overhead due to handling source code; the source file of every class must be parsed for reification and compiled for reflection. Although this overhead is compensation for the capability for fine-grained transformation of source code (including syntax extension), it is not negligible if OpenJava is used by a class loader for altering a loaded class. Some kinds of applications such as a mobile agent system do not need fine-grained transformation but fast class loading.

Although the implementations of OpenJava or Javassist have not been tuned up, the performance difference between OpenJava and Javassist is notable with respect to reification and reflection. If a class loader can be implemented with either OpenJava or Javassist, Javassist achieves shorter loading time. To show this performance difference, we compared Javassist and OpenJava with two small applications. We implemented BCA [5] and behavioral reflection presented in Section 4 with both Javassist and OpenJava and we measured the time needed for altering a given class with each implementation. For fair comparison, the implementations with Javassist write modified class files back on a local disk.

Table 5 lists the results. The execution time is the average of five continuous repetitions, which do not include the first repetition. Since a program is gradually loaded into the JVM during the first repetition, the first one is tremendously slow. For compiling a modified source file, OpenJava uses a compiler provided by the Sun JDK for Solaris. However, it never uses the `javac` command since it starts the compiler in a separate process; instead, it directly runs the compiler (`sun.tools.javac`) on the same JVM.

Although the sizes of the programs implementing the applications are almost equal between Javassist and OpenJava, Javassist processed a class more than ten times faster than OpenJava. Note that the execution time by Javassist is shorter than the time needed only for compiling a modified source file. This is because

[5] Of course, the implementation of BCA with OpenJava does not modify a class file in binary form. It emulates equivalent adaptation at source-code level.

Table 5. Performance comparison between Javassist and OpenJava

		execution time (msec)	program size (lines)	original source (lines)	original class file (bytes)	modified class file (bytes)
BCA	Javassist	42	26	24	372	551
	OpenJava	543 (172†)	17	24		548
Reflection	Javassist	142	205	35	946	3932
	OpenJava	4108 (302†)	247	35		2244

Sun JDK 1.2.2 (HotSpot VM 1.0.1), UltraSPARC II 300MHz
†compilation time by `sun.tools.javac` (Java compiler).

Javassist can move compilation penalties to an earlier stage. Even a method body is not compiled while Javassist is running; it is pre-compiled in advance and the resulting bytecode is directly copied to a target class at run time.

Bytecode translators: Bytecode translators such as JOIE [5] and the JavaClass API [7] provide a functionality similar to Javassist. They enable a Java program to alter a class definition at load time. However, they are toolkits for directly dealing with bytecode, that is, the raw data structure of a class file. For example, classes included in JOIE are ClassInfo, Code, and Instruction. They show that JOIE was designed for experienced programmers who have a deep understanding of the Java bytecode and want to implement complex transformation. On the other hand, Javassist was designed to be easy to use; it does not require programmers to have knowledge of the Java bytecode but instead it provides source-level abstraction for manipulating bytecode in a relatively safe manner. Although a range of instrumentation of a method body is restricted, we showed that Javassist can be used to implement non-trivial applications. Javassist can be regarded as a front end for easily and safely using a bytecode translator like JOIE; it is not a replacement of the bytecode translators.

Using bytecode instrumentation for implementing a reflective facility is a known technique in Smalltalk [1]. A uniqueness of Javassist against this is the design of the API providing source-level abstraction. The Javassist API was carefully designed to avoid wrongly producing a class definition containing type incorrectness.

Others: OpenJIT [18] is a just-in-time compiler that allows a Java program to control how bytecode are compiled into native code. It provides better flexibility than Javassist with respect to instrumenting a method body while OpenJIT does not allow to add a new method or field to a class. However, using OpenJIT is more difficult than using Javassist because OpenJIT requires programmers to have knowledge of both the Java bytecode and native code. Although OpenJIT can be used without knowledge of the Java bytecode if programmers use a mechanism of OpenJIT for translating bytecode into a parse tree of an equivalent Java program, overheads due to that translation has not been reported.

The idea of enabling reflection only at load time for avoiding performance problems is found in the CLOS MOP [14]. For example, the CLOS MOP allows a program to alter the algorithm of determining the super classes of a given class but the super classes are statically determined when the class is loaded; the program cannot dynamically change the super classes at runtime.

Some readers may think that Javassist is very similar to BCA. However, Javassist was designed for a wider range of applications than BCA, which is specialized for on-line class adaptation. BCA only allows to modify a given class but not to dynamically define a new class from scratch. On the other hand, BCA allows programmers to describe the algorithm of adaptation in declarative form.

6 Conclusion

This paper presented Javassist, which is an extension to the Java reflection API. Unlike other extensions, it enables structural reflection in Java; it allows a program to alter a given class definition and to dynamically define a new class. A number of language extensions are more easily implemented with structural reflection than with behavioral reflection.

For avoiding portability and performance problems, the design of Javassist is based on our new architecture for structural reflection. Javassist performs structural reflection by instrumenting bytecode of a loaded class. Therefore, it can be used with a standard JVM and compiler although structural reflection is allowed only before a class is loaded into the JVM, that is, at load time. Since a standard JVM is used, the classes processed by Javassist are subject to the bytecode verifier and the SecurityManager of Java. Javassist never breaks security guarantees given by Java.

The followings are important features of Javassist:

– Javassist is portable. It is implemented in only Java without native methods and it runs with a standard JVM. It does not need a platform-dependent class library. Portability is significant in Java programming.
– Javassist provides source-level abstraction for manipulating bytecode in a safe manner while bytecode translators, such as JOIE [5] and the JavaClass API [7], provide no higher-level abstraction. The users of Javassist do not have to have a deep understanding of the Java bytecode or to be careful for avoiding wrongly making an invalid class rejected by the bytecode verifier.
– Javassist never needs source code whereas OpenJava [20], which is another system for structural reflection with source-level abstraction, does. Since OpenJava performs structural reflection by transforming source code, it must parse and compile source code for reifying and reflecting a class. Thus a class loader using Javassist can load a class faster than one using OpenJava. However, OpenJava enables fine-grained manipulation of class definitions so that the resulting definitions may be smaller and more efficient than ones by Javassist.

The architecture that we designed for Javassist can be applied to other object-oriented languages if a compiled binary program includes enough symbolic information to construct a class object. However, the API must be individually designed for each language so that it allows a program to alter class definitions in a safe manner with respect to the semantics of that language.

Acknowledgment. The author thanks Michiaki Tatsubori, who wrote programs using OpenJava for the experiment in Section 5. He also thanks Hidehiko Masuhara for his comments on an early draft of this paper, and the anonymous reviewers for their helpful comments.

References

1. Brant, J., B. Foote, R. E. Johnson, and D. Roberts, "Wrappers to the Rescure," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 396–417, Springer, 1998.
2. Braux, M. and J. Noyé, "Towards Partially Evaluating Reflection in Java," in *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, SIGPLAN Notices vol. 34, no. 11, pp. 2–11, ACM, 1999.
3. Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, no. 10, pp. 285–299, ACM, 1995.
4. Chiba, S. and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.
5. Cohen, G. A., J. S. Chase, and D. L. Kaminsky, "Automatic Program Transformation with JOIE," in *USENIX Annual Technical Conference '98*, 1998.
6. Cointe, P., "Metaclasses are first class: The ObjVlisp model," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156–167, 1987.
7. Dahm, M., "Byte Code Engineering with the JavaClass API," Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.
8. Ellis, M. and B. Stroustrup, eds., *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.
9. Fabre, J. C. and T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.
10. Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
11. Golm, M. and J. Kleinöder, "Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible," in *Proc. of Reflection '99*, LNCS 1616, pp. 22–39, Springer, 1999.
12. Hölzle, U. and D. Ungar, "A Third Generation Self Implementation: Reconciling Responsiveness with Performance," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 29, no. 10, pp. 229–243, 1994.
13. Keller, R. and U. Hölzle, "Binary Component Adaptation," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 307–329, Springer, 1998.

14. Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol.* The MIT Press, 1991.
15. Kirby, G., R. Morrison, and D. Stemple, "Linguistic Reflection in Java," *Software – Practice and Experience*, vol. 28, no. 10, pp. 1045–1077, 1998.
16. Kleinöder, J. and M. Golm, "MetaJava: An Efficient Run-Time Meta Architecture for Java," in *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*, IEEE, 1996.
17. Masuhara, H. and A. Yonezawa, "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Languages," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 418–439, Springer, 1998.
18. Ogawa, H., K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura, "OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java," in *Proc. of ECOOP'2000*, Springer Verlag, 2000. To appear.
19. Sun Microsystems, "Java^{TM} 2 SDK Documentation." version 1.3 (beta release), 1999.
20. Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-based Macro System for Java," in *Reflection and Software Engineering* (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), LNCS 1826, Springer Verlag, 2000.
21. Welch, I. and R. Stroud, "From Dalang to Kava — The Evolution of a Reflective Java Extension," in *Proc. of Reflection '99*, LNCS 1616, pp. 2–21, Springer, 1999.
22. Wu, Z., "Reflective Java and A Reflective-Component-Based Transaction Architecture," in *Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (J.-C. Fabre and S. Chiba, eds.), 1998.