Program Transformation with Reflective and Aspect-Oriented Programming

Shigeru Chiba

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology, Japan

Abstract. A meta-programming technique known as reflection can be regarded as a sophisticated programming interface for program transformation. It allows software developers to implement various useful program transformation without serious efforts. Although the range of program transformation enabled by reflection is quite restricted, it covers a large number of interesting applications. In particular, several non-functional concerns found in web-application software, such as distribution and persistence, can be implemented with program transformation by reflection. Furthermore, a recently emerging technology known as aspect-oriented programming (AOP) provides better and easier programming interface than program transformation does. One of the roots of AOP is reflection and thus this technology can be regarded as an advanced version of reflection. In this tutorial, we will discuss basic concepts of reflection, such as compile-time reflection and runtime reflection, and its implementation techniques. The tutorial will also cover connection between reflection and aspect-oriented programming.

1 Introduction

One of significant techniques of modern software development is to use an application framework, which is a component library that provides basic functionality for some particular application domain. If software developers use such an application framework, they can build their applications by implementing only the components intrinsic to the applications. However, this approach brings hidden costs; the developers must follow the protocol of the application framework when they implement the components intrinsic to their applications. Learning this framework protocol is often a serious burden of software developers. A richer application framework provides a more complicated protocol. Learning the protocol is like learning a new domain-specific language since software developers must understand the programming model or the underlying architecture.

A program translator has a potential ability to simplify such a framework protocol. If framework developers use a program translator, they can provide a simpler but not-real framework protocol for the users. The users can implement their application-specific components with that simple protocol. Then they can translate their components by the program translator into ones following a complex but real protocol of the application framework. Their translated components can actually run with the application framework. A challenge of this idea is to make it easy for framework developers to implement a program translator for their framework. Since framework developers are normally experts of not compiler technology but the application domain of the framework, the easiness of developing a program translator is a crucial issue in this scenario. Furthermore, if the development cost of a program translator is extremely high, it would not be paid off by the benefits of the simple protocol.

Reflection [28, 27, 21] can be regarded as one of the technology for making it easy to implement such a program translator for simplifying a framework protocol. The power of meta programming by this technology allows framework developers to implement a simple protocol without directly developing a program translator for their framework. The framework developers can describe the algorithm of program transformation with high-level abstraction and the actual program transformation is implicitly executed by the underlying mechanism for reflective computing. They do not have to consider source-code parsing, semantic analysis, or other messy details of implementation of a program translator. Although reflection used to be known as the mechanism involving serious performance overheads, today there are several implementation techniques for efficient reflective computing. Carefully designed reflective computing does imply zero or only a small amount of overhead.

Aspect-oriented programming (AOP) [19] is relatively new technology; as well as reflection, it is useful for simplifying a framework protocol. One of the roots of AOP is the study of reflection and thus AOP is often regarded as a descendant of the reflection technology. AOP is not another new name of reflective computing or technology hype. AOP is new technology for reducing dependency among software components. If components have strong dependency on each other, the protocol of interaction among the components will be complicated and difficult to learn. AOP reduces the dependency among components, in particular, between components implemented by application developers and components provided by an application framework. Reducing the dependency makes the framework protocol simpler and easier to use. AOP does not directly help implementing a program translator but it is a programming paradigm that provides language constructs for simplifying a framework protocol. The ability of those constructs are quite equivalent to what we want to achieve by using a program translator.

In the rest of this paper, we first discuss how program translators can simplify the protocols of application frameworks. Then this paper presents overviews of the reflection technology. It also describes AOP. We discuss what are benefits of AOP and the unique functionality of AOP, which is not provided by the reflection technology.

2 Program Translator

To illustrate our motivation, this section first presents the complexity of the protocols of application frameworks. Then it mentions how a program translator can simplify the protocols.

2.1 Simple example: Graphical User Interface library

Application frameworks are class libraries that can be a platform for building application software in a particular application domain. Since they provide basic building blocks in that domain, they significantly improve efficiency of software development. However, to exploit the power of application frameworks, application developers must learn the protocols of the frameworks. Unfortunately, those protocols are often complicated and thus learning the protocols is often a time consuming job. It is hidden cost of software development with application frameworks.

The complications of a framework protocol mainly come from the limited ability of programming languages to modularize software. For example, we below show a (pseudo) Java program written with the standard GUI framework. It is a program for showing a clock. If this program does not have GUI, then it would be something like the following simple and straightforward one (for clarifying the argument, the programs shown below are pseudo code):

```
class Clock {
  static void main(String[] args) {
    while (true) {
      System.out.println(currentTime());
      sleep(ONE_MINUTE);
    }
  }
}
```

This program only prints the current time on the console every one minute.

Now we use the standard GUI library as an application framework to extend this program to have better look. To do that, however, we must read some tutorial books of the GUI library and edit the program above to fit the protocol that the books tell us. First, we would find that the Clock class must extend Panel. Also, the Clock class must prepare a paint method for drawing a picture of clock on the screen. Thus you would define the paint method and modify the main method. The main method must call not the paint method but the repaint method, which the tutorial book tells us to call when the picture is updated. The following is the resulting program (again, it is pseudo code):

```
class Clock extends Panel {
  void paint(Graphics g) {
    // draw a clock on the screen.
  }
  static void main(String[] args) {
    Clock c = new Clock();
    while (true) {
      c.repaint();
      sleep(ONE_MINUTE);
    }
  }
}
```

Note that the structure of the program is far different from that of the original program. It is never simple or straightforward. For example, why do we have to define the paint method, which dedicates only to drawing a picture? Why does the main method have to call not the paint method but the repaint method, which will indirectly call the paint method? To answer these questions, we have to understand the programming model or the underlying architecture of the framework provided by the GUI library. This is hidden cost that application developers have to pay for exploiting the power of application frameworks.

2.2 Enterprise Java Beans

Enterprise Java Beans (EJB) [29] is a popular application framework for building a web application in Java. This framework provides basic functionality such as transaction, distribution, and security, for software components developed by the users. Since the cost of implementing such functionality for each component is never negligible, the use of EJB significantly reduces the development cost of application software.

However, the benefits of EJB are not free. The developers must follow the protocol of EJB when they develop their software components. This protocol (at least, before EJB 3.0) is fairly complicated whereas the power of the EJB framework is strong. Only the developers who spend their time on mastering the framework protocol of EJB can enjoy the power of EJB. For example, to implement a single EJB component, the developer must define three classes and interfaces in Java. Suppose that we want to implement a component for dealing with registration of summer school. Ideally, the definition of this component would be something like this:

```
class Registration {
  void register(String who) { ... }
  void cancel(String who) { ... }
}
```

However, to make this component be EJB-compliant, we must define the following class and interfaces (again, these are pseudo-code):

```
class RegistrationBean implements SessionBean {
  void register(String who) { ... }
  void cancel(String who) { ... }
}

interface RegistrationHome {
  Registration create() throws .. ;
}

interface Registration extends EJBObject {
  void register(String who) throws .. ;
  void cancel(String who) throws .. ;
}
```

Note that Registration is now the name of the interface extending EJBObject although it was the class name in the ideal version. The name of the class implementing the component functionality is now RegistrationBean. The protocol is not only the names; the reason why we must define extra interfaces is not clear unless we know the underlying architecture of EJB.

2.3 Use of Program Translators

The complications of framework protocols we showed above can be simplified if the application frameworks are distributed with custom program translators. The framework designers can define a simple but not-real protocol, with which the framework users implement their components. These components *as is* cannot work with the application frameworks but they can be translated into the components that follow the real protocols of the application frameworks. In fact, this idea has been adopted by upcoming EJB 3.0. EJB 3.0 provides an extremely simple protocol; it allows developers to implement EJB components as regular Java objects. The developers do not have to follow any protocol when they implement EJB components. So the developers can write the ideal code we have seen above:

```
@Session class Registration {
  void register(String who) { ... }
  void cancel(String who) { ... }
}
```

The components described as above are translated by a sort of program translator into the components that follows the real but hidden protocol of the framework so that they can work with the framework.

However, this approach using a program translator has not been widely used yet. A main reason would be that implementing a program translator is not easy. Only application frameworks that have a large user base can adopt this approach since the cost of implementing a program translator is paid off by the benefits that the users can receive. EJB is a typical example. To make this approach widely used for simplifying a framework protocol, we have to study technologies for easily implementing a program translator customized for each application framework. One of such technologies is reflection and another is aspect-oriented programming (AOP). In the following sections, we discuss these two technologies.

3 Reflection

Reflection, or reflective computing, is one of meta-programming technologies originally proposed by Brian C. Smith in the early 1980's [28, 27]. He presented this idea by showing his Lisp language called *3-Lisp*. This idea was extended and applied to object-oriented languages by Pattie Maes, who proposed the *3-KRS* language [21], and others during the 1980's. Smalltalk-80 [15] is also recognized today as one of the early reflective languages [12] although it was not intended to enable reflective computing when it was designed.

3.1 Reify and Reflect

Reflection allows a program to access the *meta-level view* of that program itself. If an entity in that meta-level view is changed, the corresponding entity in the original program is really changed. Therefore, reflection enables a program to transform (parts of) that program itself within the confines of the language. Self-reflective languages, that is, programming languages that enable reflective computing are like surgeons who can perform an operation on themselves for cancer.

The primitive operations of reflective computing are *reify* and *reflect*. Reifying is to construct a data structure representing some non-first-class entity in a program. It is also called *introspection*. The metaobject can be queried for the structure of the represented entity. For example, the standard reflection API of Java provides a Class object representing a class. Such an object as the Class object is often called a *metaobject* since it is part of the meta-level view of a program. The Class object can be queried for its super class, methods, and so on. Constructing the Class object is a typical reifying operation since a class is a non-first-class entity in Java. That is, the forName method in Class is the reifying operation provided by the reflection API of Java. Here, the first-class entity means a data value that a program can deal with. The non-first-class entities are the rest of the entities included in a program. For example, an integer and a String object are first-class entities. On the other hand, a class and a method are non-first-class entities since variables cannot hold them or refer to them.

Reflecting is to alter the structure or the behavior of a program according to the changes applied to the metaobjects (or some data structure representing the meta-level view if the language is not object-oriented) obtained by the reifying operation. Note that a metaobject is a regular object although it represents part of the meta-level view of a program. Therefore, it is not obvious that changes in a metaobject are reflected on the structure or the behavior of the program. To emphasize that the changes are reflected on the program, a reflective language has *causal connection* between a program and metaobjects (or data structures representing the meta-level view). The reflection API of Java does not provide this type of reflecting operation. The metaobjects such as Class, Method, and Field objects provide only getter methods but not setter methods. Thus, any changes cannot be applied to the metaobjects.

Another type of reflecting is to execute base-level operations, such as object creation and method invocation, through a metaobject. The reflection API of Java provides this type of reflecting operation. The Class class includes the newInstance method, which makes an instance of the class represented by the Class object. The Method class includes the invoke method, which invokes the method. The computation by these methods is also the reflecting operation since it is reflected on the real program execution.

3.2 Metaobject Protocol

CLOS [1] is known as an object-oriented language that has strong ability with respect to both reifying and reflecting operations. The reflection API of CLOS is called the CLOS *Metaobject Protocol* (MOP) [18]. CLOS is an object system built on top of Common Lisp and thus an object in CLOS is implemented by using a data structure of Common Lisp, for example, an array. If a class is defined in CLOS, an object representing a class is created at runtime. This object contains the information of the class definition. Note that this fact does not cause infinite regression. From the implementation viewpoint, this fact just means that the runtime data structure representing a class is the same data structure of Common Lisp that is used for representing an object. However, we interpret this fact as that a class is an object in CLOS as we say that a class is an object in Smalltalk-80.

The class definition in CLOS is expanded by a macro into an expression that makes an object. Recall that Lisp macros are powerful programmable macros. This architecture of CLOS is illustrated by the following pseudo Java code:[1]

```
public class Point {
  int x, y;
  void move(int newX, int newY) { ... }
}
```

This class definition is expanded by a macro into the following expression:

```
Class pointClass
  = new Class("Point",
              new Field[] { new Field("int", "x"),
                            new Field("int", "x") },
              new Method[] { new Method("void", "move", ... ) });
```

Here, pointClass is sort of a global variable. For each class definition, a Class object is created at runtime and it contains the information about the class definition. Note that there is no syntactical distinction among expressions, statements, and declarations in Lisp. Thus transforming a class declaration into an expression as shown above is valid macro expansion in Lisp.

An expression for making an instance of Point is also expanded by a macro. For example, this expression:

```
Point p = new Point();
```

is transformed into this:

```
Point p = (Point)pointClass.newInstance();
```

newInstance declared in the Class class is a method for making an instance. Furthermore, the following expression for calling a method in Point:

```
p.move(3, 4)
```

is transformed into something like this:

```
pointClass.getMethod("move").invoke(p, new Object[] { 3, 4 });
```

[1] The object model of CLOS is quite different from that of Java. So the following explanation is not exactly about the CLOS MOP. We try to show the basic idea of the CLOS MOP with the context of Java since most of the readers would be familiar to Java.

The resulting expression first obtains a Method object representing move and then invokes the method with arguments 3 and 4. In principle, since the resulting expression consists of two method calls (getMethod and invoke), it would be also expanded by the same macro. However, this macro expansion performs different transformation; the expression is transformed into a Lisp expression implementing the normal behavior of method call. It does not include a method call or any other object-oriented operators. If the macro expansion were naively applied, it would cause infinite regression.

As we showed above, all the operations related to objects in CLOS are always transformed into method calls to Class objects, at least, in the programming model of CLOS. Therefore, if we call a method on the Class objects and change the states of those objects, the changes are immediately *reflected* on the behavior of the operations related to objects.

For example, we can dynamically add a new method to the Point class by explicitly calling a method on the Class object representing the Point class:

```
pointClass.addMethod(new Method("String", "toString", ...));
```

This expression first makes an instance of Method that represents a toString method. Then it adds the method to the Point class by calling the addMethod method.

If a subclass of Class is defined and some methods in Class is overridden, the behavior of the operations such as object creation and method calls can be altered. This alteration is called *intercession*. For example, let us define the following subclass:

```
public class TracedClass extends Class {
  public Object newInstance() {
    System.out.println("instantiate " + getName());
    return super.newInstance();
  }
}
```

Then define a Point class as following:

```
public class Point is_instance_of TracedClass {
  int x, y;
  void move(int newX, int newY) { ... }
}
```

Here is_instance_of is a keyword for specifying the class of the class metaobject. This class definition is expanded by the macro into the following statement:

```
Class pointClass = new TracedClass("Point", ... );
```

Now the class metaobject that pointClass refers to is an instance of TracedClass. Hence, if an instance of Point is made, a trace message is printed out.

3.3 Operator Interception

Reflection enables developers to define a method that is executed when an operator such as method call and field access is executed. If the thread of control reaches such an operator, the program execution is intercepted and then that defined method is invoked instead of the operator. The TracedClass example shown above is an example of such an intercepting method. We defined a method intercepting a new operator in the TracedClass class.

Although such an intercepting method has a large number of practical applications and it is provided by most of reflective languages, an intercepting method is not a unique mechanism of the reflection technology. It is also provided by other technologies such as aspect-oriented programming. A unique feature of the reflection technology is that it enables an intercepting method to access the contexts of the intercepted operator through a meta-level view so that the intercepting method could be *generic*.[2]

For example, the newInstance method in the TracedClass example can intercept the new expressions (object-creation expressions) of making an instance of any class type. It can intercept a new expression for either Point or Rectangle. This is because the newInstance method is at the meta level and hence the class type of the created object is *coerced* to the Object type. Another example is the invoke method in Method. This method receives arguments in the form of array of Object independently of the types and the number of the arguments. This enables a generic intercepting method, which can intercept different kinds of method calls. The following example illustrates the invoke method overridden in a subclass so that it will print a trace method:

```
public class TracedMethod extends Method {
  public Object invoke(Object target, Object[] args) {
    System.out.println("method call " + getName());
    return super.invoke(target, args);
  }
}
```

The invoke method is an intercepting method, which is executed when a method represented by a TracedMethod metaobject is called. This invoke method can intercept any method calls, no matter how many parameters or what type of parameters the method receives. This is because the list of arguments is *reified* by using an array of Object.

How the base-level entities such as the number and types of method arguments are reified depends on the reflective language. For example, OpenC++ (version 1) [7] uses an instance of the ArgPac class for representing the method arguments at the meta level since there is no root type of all the class types in C++.

[2] This feature is also provided by an aspect-oriented programming language AspectJ. However, it is a reflection mechanism of AspectJ according to the documents of AspectJ.

3.4 Structural Reflection v.s. Behavioral Reflection

Reflective programming is classified into two categories: structural reflection and behavioral reflection. Structural reflection is to alter a program structure, that is, a class definition in Java, through a metaobject, for example, defining a new class, adding a new method to an existing class, removing an existing field, and changing a super class. Structural reflection enables straightforward implementation of program transformation while keeping simple abstraction for describing the transformation.

Behavioral reflection is to alter the behavior of operations in a program. Typical behavioral reflection is to define a subclass of Class or Method and override methods for executing operations such as object creation, method calls, and field accesses. For example, in Section 3.3, we defined a subclass of Method for altering the behavior of method invocation to print a trace message:

```
public class TracedMethod extends Method {
  public Object invoke(Object target, Object[] args) {
    System.out.println("method call " + getName());
    return super.invoke(target, args);
  }
}
```

This is typical behavioral reflection since it alters the semantics of method invocation through a metaobject instead of transforming a program to print a trace message.

The main difference between structural reflection and behavioral reflection is a programming model. The expressive power of the two kinds of reflection is, in principle, equivalent to each other. For example, a program can be evolved to print a trace message by either structural reflection or behavioral reflection. If behavioral reflection is available, the tracing mechanism can be implemented as we have already showed above. On the other hand, if structural reflection is available, statements for printing a trace message can be embedded at appropriate places in the method bodies included in the program. A method metaobject would allow substituting the body of the method or instrumenting statements included in that body. Suppose that a Method object provides an insertBefore method, which inserts a given statement at the beginning of the method body. Then the following statement transforms the move method in the Point class so that it will print a trace message when it is called:

```
pointClass.getMethod("move")
          .insertBefore("System.out.Println(\"method call\");");
```

Note that the variable pointClass refers to the Class object representing the Point class.

This discussion is analogous to how to implement language extensions. If we want to extend a programming language, we have two approaches for the implementation. The first one is to implement a source-to-source program translator from the extended language to the original language. The other one is to extend a compiler or interpreter of the original language so that it can deal with the new features of the extended language. The structural reflection corresponds to the former approach while the behavioral reflection corresponds to the latter approach.

The programming model of structural reflection is easily derived from the program structure of the target language. If the target language is Java, the metaobjects are classes, methods, constructors, and fields. On the other hand, the behavioral reflection has a variety of programming models. In one programming model, the metaobjects are classes, methods, and so on. In another model, each object is associated with a metaobject representing a virtual interpreter that is responsible to the execution of the operations, such as method calls, on that object. Developers can customize that metaobject to alter the behavior of only the particular object instead of all the instances of a particular class. There is also a programming model in which a garbage collector and a thread scheduler are metaobjects. Developers can customize their behavior through the metaobjects [32, 25]. Another model uses a metaobject representing a message exchanged among objects [11] or communication channels [2].

3.5 Typical Applications

A typical application of reflective computing is to implementing a program translator. Structural reflection is obviously useful for that purpose. It allows developers to concentrate the procedure of program transformation at the source-code level. They do not have to implement a program parser or to transform an abstract syntax tree.

For example, structural reflection can be used to implement a program translator that automatically transforms a program written as a non-distributed program so that it can run on multiple hosts as a distributed program. The essence of such program transformation is to produce the class definitions for proxy objects and modify the program to substitute a reference to a proxy object for a reference to a local object when the local reference is passed to a remote host. To produce the class definitions for proxy objects, the class definitions for the original objects that the proxy objects refer to must be investigated; the names and the signatures of the methods declared in the class must be obtained. The reifying capability of metaobjects helps this investigation. Structural reflection allows developers to easily produce the class definitions for proxy objects by constructing new class metaobjects. Substituting object references can be implemented by modifying method bodies through method metaobjects.

Behavioral reflection provides a direct solution for program translators that implement some language extensions such as distribution, persistence, and transaction. It is useful to transform a program so that those non-functional concerns will be appended to classes in the program. For example, it can be used to implement the synchronized method of Java. Some languages like C++ do not provide the language mechanism of the synchronized method but it can be implemented by a customized method metaobject. If Java did not provide synchronized methods but supported behavioral reflection, the class for the customized method metaobject would be as following:

```
public class SynchronizedMethod extends Method {
  public Object invoke(Object target, Object[] args) {
    synchronized (target) {
      return super.invoke(target, args);
    }
  }
}
```

The synchronized statement locks the target while the following block statement is being executed.

If a language supporting behavioral reflection provides a metaobject representing a thread scheduler, the metaobject can be used to implement an application-specific scheduler. In the area of scientific computing with parallel processing, thread scheduling optimized for particular application software can often significantly improve the execution performance of that software [23]. Suppose that a tree data structure must be recursively searched in parallel. For some applications, the depth-first search approach should be used while for other applications the breadth-first search approach should be used. If the thread scheduler can be customized through a metaobject, the application software can adopt the most appropriate scheduling policy. In general, the maximum number of concurrent threads should be decreased for the depth-first search whereas it should be increased for the breadth-first search.

3.6 Implementation Techniques

Reflection was regarded as a mechanism that was useful but too inefficient to use for practical purposes. The most significant problem was that a language supporting reflection tended to imply a serious performance penalty even when the reflection mechanism was not used at all. If a language fully provides the reflection capability, any parts of the program structure and the behavior of any operations must be changeable through metaobjects. A naive implementation of the language processor for such a language is an interpreter and thus it cannot run the program efficiently.

To avoid this performance problem, several implementation techniques have been proposed so far. A simple technique is to restrict a kind of metaobjects available. For example, if we know in advance that only a limited number of classes are reified, the other classes that are not reified can be normally compiled into efficient binary code. Otherwise, the language processor can first run a program with the compiled binary code and then, when classes are reified, it can stop using the compiled binary code and start executing those classes with an interpreter [23].

If the reflection capability allows only intercession, efficient implementation is relatively easy. The whole program can be normally compiled into efficient binary except that *hook code* is inserted around the operators customized by metaobjects [7, 33, 16]. When the thread of control reaches one of those operators, the hook code intercepts the execution and switches the control to the metaobject, which executes extended behavior of the intercepted operator. Although the execution of the operators customized through metaobjects imply performance penalties, the execution of the rest of the operators does not involve an overhead due to intercession.

Another technique is to partly recompile a program whenever the structure or the behavior of the program is altered through a metaobject. For example, if a class definition is changed, the runtime system of the language can recompile that class definition on the fly so that the changes of the class will be reflected. Note that this technique makes it difficult to perform global optimization. Suppose that a method is inlined in other methods at compilation time. If the body of that inlined method is changed through a metaobject at runtime, all the methods where that method is being inlined must be also recompiled during runtime.

Curring and Memoizing: The CLOS MOP adopts the *curring and memoizing* technique [18] for efficient implementation. This technique is similar to the dynamic recompilation technique above. To illustrate the idea of the curring and memoizing technique, first let us consider the following naive implementation of the newInstance method (written in Java for readability). Assume that we implement an object as an array of Object:

```
public class Class {
  public Object newInstance() {
    int size = getSuperclass().computeFieldSize();
    size += computeFieldSize();
    return new Object[size];
  }
        :
}
```

Although newInstance in Java is a native method, that method is a regular method in the CLOS MOP so that it can be customized by subclassing the Class class.

Unfortunately, this implementation is too slow since it repeatedly computes the object size whenever a new instance is created. To avoid this inefficiency, the curring and memoizing technique uses a function closure. See the improved version of newInstance below:

```
public class Class {
  private Closure factory;
  public Object newInstance() {
    if (factory == null)
      factory = getFactory();
    return factory.newInstance();
  }
  public Closure getFactory() {
    int s = getSuperclass().computeFieldSize();
    final int size = s + computeFieldSize();
    return new Closure() {
      public Object newInstance() {
        return new Object[size];  // size is constant
      }
    };
  }
      :
}
```

The getFactory method is a sort of compiler since compilation is a process of transforming a program to an efficient form by using statically available information. Indeed, that method transfomrs a new expression and returns a function closure[3] that efficiently makes an instance. Note that the computeFieldSize method is called only once when the closure is created. The returned closure is memoized in the factory field to avoid redundantly calling getFactory. From the next time, the newInstance method creates an instance by calling this function closure.

Now let us define a subclass of Class so that a trace message is always printed out when a new object is created. We override the getFactory method:

```
public class TracedClass extends Class {
  public Closure getFactory() {
    final Closure c = super.getFactory();
    final String name = getName();
    return new Closure() {
      public Object newInstance() {
        System.out.println("instantiate " + name);
        return c.newInstance();
      }
    };
  }
}
```

Note that super.getFactory() and getName() are called only once when the closure is created.

In the CLOS MOP, a subclass of Class does not directly specify how to create an instance. Rather, it specifies how to construct an efficient function for creating an instance. Hence the class metaobject of that subclass can be regarded as a compiler. The runtime system of the CLOS MOP calls that metaobject to obtain the "compiled code" and then continues to use it until another reflecting operation is performed.

The CLOS MOP requires metaobject developers to describe how to construct an efficient function for doing basic operations such as object creation. Hence the programming style is not simple or straightforward although the runtime overhead is low.

Partial Evaluation: A number of applications do not need full reflection capability. They need the reflecting operation only at the beginning of the program execution. Once the reflecting operation is performed, they do not need to alter program structure or behavior again during runtime. In fact, altering program

[3] In the CLOS MOP, getFactory really returns a closure.

structure during runtime is not a simple job. For example, if a new field is added to an existing class, we must also specify how the existing instances of that class are dealt with. Is the new field also added to those existing instances? If so, what is the initial value of that added field?

If a program does not need the full reflection capability and some reflecting operations are statically determined, we can compile the program into efficient code. A few researchers [22, 24] have proposed using the technique called *partial evaluation* for compiling away metaobjects. Their idea is to apply partial evaluation [10, 14] to the class definitions for metaobjects. The static input for the partial evaluation is the program running with the metaobjects.

From a pragmatic viewpoint, partial evaluation is a theoretical framework for agressively performing constant propagation and code inlining. We below illustrate the basic idea of partial evaluation from the pragmatic viewpoint. For example, suppose we have the following simple class definition for metaobjects:

```
public class TracedClass extends Class {
  public Object newInstance() {
    System.out.println("instantiate " + getName());
    return super.newInstance();
  }
}
```

Then suppose that a program includes the following statement:

```
Point p = new Point();
```

This statement is first translated into the following statement calling a metaobject as in the CLOS MOP:

```
Point p = (Point)pointClass.newInstance();
```

pointClass refers to the class metaobject representing the Point class. Then the partial evaluator inlines the newInstance method in TracedClass:

```
System.out.println("instantiate " + pointClass.getName());
Point p = (Point)pointClass.super.newInstance();
```

Next, let us inline getName() and newInstance():

```
System.out.println("instantiate Point");
int size = pointClass.getSuperclass().computeFieldSize();
size += pointClass.computeFieldSize();
Point p = (Point)new Object[size];
```

Since the resulting values of getSuperclass() and computeFieldSize() are constant, those method calls turn into constants. Let the value of size be 5. Thus the resulting statement is as following:

```
System.out.println("instantiate Point");
Point p = (Point)new Object[5];
```

This statement is the result of partial evaluation but it does not include any calls to the metaobject.

Although partial evaluation is a powerful technique for efficiently implementing the reflection mechanism, developing a partial evaluator is extremely difficult. The articles [22, 24] reported that the authors succeeded in developing a partial evaluator for their Lisp-based ABCL/R3 reflective language. Unfortunately, no article has yet reported that a partial evaluator can be used to efficiently compile reflective computing in C++ or Java as far as the author knows.

Compile-time reflection If the reflection capability is necessary only at the beginning of program execution and not necessary during runtime, we can use another compilation approach, which is called *compile-time reflection*. It was developed by the author's group for reflective languages OpenC++ [3, 4] and OpenJava [31]. The compile-time reflection allows reflective computing only at compile time — therefore, a code block performing reflective computing must be separated from the rest of the program, which performs normal base-level computing. The *meta* program performing reflective computing is executed by a compiler at an early stage of the compilation process as macro expansion is.

The meta program can directly describe program transformation. Suppose that pointClass refers to the metaobject representing the Point class. If the meta program includes the following statement:

```
pointClass.addField(new Field(intType, "z"));
```

this meta program is separately compiled in advance and then executed at the compile time of the target (base-level) program. If executed, the meta program appends a new int field named z to the Point class. After that, the resulting target program is normally compiled into binary code.

An advantage of the compile-time reflection is that the overhead due to reflective computing is negligible or zero. To implement a program translator, the compile-time reflection is an appropriate tool since it provides high-level abstraction, such as class metaobjects and method metaobjects, with negligible overheads. The developers do not have to directly manipulate an abstract syntax tree for program transformation.

The meta program for the compile-time reflection can also include user-defined classes for metaobjects. However, as the metaobject of the CLOS MOP returns a function closure, the metaobject of the compile-time reflection returns transformed source code. For example, see the following class:

```
public class TracedClass extends Class {
  public String newInstance() {
    return "(System.out.println(\"instantiate " + getName() + "\"),"
           + super.newInstance() + ")";
  }
}
```

The compiler for the compile-time reflection first reads a target program and, when it finds an expression for object creation, method call, or field access, it queries the corresponding metaobject about how the expression should be transformed. The compiler uses the static type of the expression for selecting the metaobject. Then the compiler replaces the original code with the source code returned from the metaobject. After finishing the replacement of all the expressions, the compiler compiles the program into binary code.

For example, suppose that a target program includes the following statement:

```
Point p = new Point();
```

and the class metaobject for Point is an instance of TracedClass. When the compiler finds this object creation, it calls the newInstance method in TracedClass and requests it to transform "new Point()". Since the newInstance method in Class returns "new Point()" (*i.e.* no transformation), the newInstance method in TracedClass returns the following code:

```
(System.out.println("instantiate Point"), new Point())
```

Note that we use for readability the comma (,) operator of C++. If an expression "(*expr1*, *expr2*)" is evaluated, *expr1* and *expr2* are evaluated in this order and the resulting value of *exp2* becomes the resulting value of the whole expression. The returned code above is substituted for the original expression "new Point()". After this substitution, the target program becomes:

```
Point p
  = (System.out.println("instantiate Point"), new Point());
```

Finally, this program is normally compiled. The resulting binary code will not include any metaobjects.

Load-time Reflection: We have also developed a variant of the compile-time reflection. It is *load-time reflection* and it was adopted for designing our Java bytecode engineering toolkit named *Javassist* [5, 8].

The load-time reflection allows reflective computing only at load time whereas the compile-time reflection allows at compile time. Both types of reflection do not allow reflective computing during runtime. From the programming viewpoint, there is no differences between the two types of reflection. However, from the implementation viewpoint, there is a significant difference between them: the target program of the compile-time reflection is source code but that of the load-time reflection is compiled binary.

OpenC++, which is based on the compile-time reflection, reads source code and constructs metaobjects. Changes to the metaobjects are reflected back to the source code before compilation. On the other hand, Javassist reads a Java class file (*i.e.* Java bytecode obtained after source code is compiled) and constructs metaobjects. Changes to the metaobjects are reflected on the class files before the JVM (Java Virtual Machine) loads them. This architecture is possible at least in Java since Java class files contain rich symbol information.

Note that the users of Javassist does not have to learn the internal structure of Java class files or the instructions of Java bytecode. They can enjoy the source-level abstraction provided by metaobjects as they can in OpenC++. The changes to the metaobjects are described with Java source code and, when they are reflected on the class files, Javassist automatically translates them into the changes described at bytecode level. For example, the following meta program appends a new method toString to the Point class:[4]

```
Method m = new Method(
  "public String toString() { return this.x + \",\" + this.y; }");
pointClass.addMethod(m);
```

pointClass refers to the class metaobject representing the Point class. Note that the method body is given in the form of Java source code. This source code is compiled by the internal compiler of Javassist and then embedded in a class file. The users do not have to construct a sequence of Java bytecode for specifying a method body. Providing source-level abstraction is an advantage of Javassist against other naive bytecode engineering toolkits such as BCEL [9].

4 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [19] is an emerging paradigm for modularizing a *crosscutting concern*, which is strongly relevant to other concerns and thus cannot be implemented as an independent component or module in a traditional language. The implementation of a crosscutting concern in a traditional language, for example, an object-oriented language like Java, often consists of not only a normal independent component but also code snippets spread over the components implementing other concerns. Therefore, such an implementation of a crosscutting concern is difficult to append and remove to/from software without editing the implementations of other concerns when the requirements of the software are changed. The goal of aspect-oriented programming is to provide language mechanisms to solve this problem.

From the viewpoint of program transformation, AOP languages such as AspectJ [20] provide several useful mechanisms for more simply achieving the goals that we have been doing with typical program transformation tools or reflection mechanisms. In fact, one of the roots of AOP is the reflection technology. Describing rules or algorithms for program transformation is never easy but it is often error-prone. For example, one algorithm for program transformation might be not general enough to cover all possible programs and thus it might not work if a particular program is given as the input. AOP languages like AspectJ provides mechanisms integrated into the language semantics and thereby they let us avoid directly describing error-prone rules for program transformation.

4.1 AspectJ

To illustrate the overview of AOP, we show a simple program written in AspectJ. A famous example of AOP is logging, which is a typical crosscutting concern.

[4] This is not a real program for Javassist.

Suppose that we want to print a logging message when the paint method is called.

Example: The main body of the implementation of the logging concern can be modularized into a single class, for example, in Java:

```
public class Logging {
  private PrintStream output = System.out;
  public static void setStream(PrintStream out) {
    output = out;
  }
  public static void print(String m) {
    output.println(m);
  }
}
```

This component encapsulates which output device is used for printing a logging message.

The rest of the work is to edit the paint method so that the print method in Logging will be called:

```
public class Clock extends Panel {
  public void paint(Graphics g) {
    Logging.print("** call paint method");    // edit!
    // draw a clock on the screen.
  }
  public static void main(String[] args) { .. }
}
```

Although this is a very typical Java program, the implementation of the logging concern cuts across other components such as Clock. It is not an independent component separated from other components. Some of the readers might think that the implementation of the logging concern is separated into the Logging class. However, that thought is wrong since the implementation of the logging concern also include the expression for calling the print method. This caller-side expression specifies when a logging message is printed out. The argument of the call specifies the contents of the printed message. Although these issues are part of the logging concern, that expression is not encapsulated in Logging but embedded in the paint method in Clock.

Using an Aspect: An AspectJ, the logging concern can be implemented as a single independent module called an *aspect*. See the following program:

```
aspect Logging {
  private PrintStream output = System.out;
  public void setStream(PrintStream out) {
    output = out;
  }
  public void print(String m) {
    output.println(m);
  }
  // before advice
  before(): call(void Clock.paint(Graphics)) {
    print("** call paint method");
  }
}
```

Note that Logging is now not a class but an aspect and it includes before advice:

```
before(): call(void Clock.paint(Graphics)) {
  print("** call paint method");
}
```

This specifies that the print method is called just before the paint method in Clock is called. An instance of the Logging aspect is a singleton and automatically created. The AspectJ compiler automatically modifies the definition of the Clock class to implement this behavior. Thus the developer can write the Clock class without considering the logging concern:

```
public class Clock extends Panel {
  public void paint(Graphics g) {
    // draw a clock on the screen.
  }
  public static void main(String[] args) { .. }
}
```

The paint method does not include an expression for calling the print method in Logging. Hence Logging and Clock are completely separated from each other. They can be combined and disconnected on demand without editing the source code of those components.

Joinpoints The key concepts of AOP is joinpoints, pointcuts, and advice. In this programming paradigm, program execution is modeled as a sequence of fine-grained events, such as method calls, field accesses, object creation, and so on. These events are called *joinpoints. pointcuts* are filters of joinpoints. They select interesting joinpoints during program execution. Then, if a joinpoint selected by some pointcut occurs, the *advice* associated to that pointcut is executed. In the case of the example above,

```
call(void Clock.paint(Graphics))
```

is a pointcut. It specifies method calls to the paint method in Clock. call is one of the primitive pointcut designators provided by AspectJ. There are various pointcut designators for selecting different kinds of joinpoints. The advice is the declaration beginning with before and ending with a block `{..}`.

A crosscutting concern is implemented as a set of advice in an aspect. The connection between the aspect and other classes is described by pointcuts. Joinpoints can be regarded as execution points at which an aspect and a class are connected to each other.

Intertype declaration An aspect of AspectJ may contain an intertype declaration as well as regular methods and fields, pointcuts, and advice. The intertype declaration declares a method or a field in another class. Developers can use intertype declarations for appending methods and fields to existing classes. For example, the following aspect appends the paintCount field to the Clock class:

```
aspect Counter {
  int Clock.paintCount = 0;
}
```

4.2 Dependency Reduction

AOP is a technology for implementing a crosscutting concern as a separate component. A component implementing such a concern in a traditional language is tightly coupled with other components so that it cannot be treated as an independent component. It cannot be disconnected from the other components when it is not necessary any more, or it cannot be connected to the existing software on demand. AOP languages provides language mechanisms for loosing this dependency among components so that the components can be treated as an independent one. Those components independently described are connected to each other at compile time or runtime to constitute complete software. This connecting process is called *weaving.*

This weaving process is similar to the transformation by a number of program translators for simplifying a framework protocol. Their role is also connecting user components to framework components. This is why AOP can be a technology for simplifying a framework protocol.

The language mechanisms of AspectJ are similar to the ones for reflective computing but they are carefully designed so that developers can easily implement a crosscutting concern. A disadvantage of reflective computing is that a program for reflective computing is often complicated and difficult to maintain. The reflective computing is powerful enough to write a program that lets you *really* shoot yourself in the foot, that is, break the program itself. For example, it allows developers to wrongly remove a method that is called by other methods. It allows them to change the super class of a class to an irrelevant class. The language mechanisms of AspectJ are less powerful but relatively safe and easy to use.

GluonJ: Several AOP languages have extended the language design of AspectJ so that they can support applications that is difficult for reflective computing to cover. Our AOP language named *GluonJ* [6] provides a stronger language mechanism for separating components. GluonJ enables reducing dependency among components for the separation. In other words, it provides powerful mechanisms for connecting components that are described independently of each other.

GluonJ is an AOP extension to Java, which is mainly for addressing an issue similar to the issue that the dependency injection [13] is trying to address. The dependency injection is a popular feature of recent component frameworks for web applications. It improves the reusability of the components built on top of other lower-level sub-components. Suppose that an component contains other components. The idea of the dependency injection is that, when a factory object creates a component, it also creates the sub components contained in that component and automatically sets the fields of that component to those sub components. This makes the source code of the component independent of the code for creating the sub components. For example, a component implementing business logic may contain a sub component for accessing a database:

```
class Registration {
  DbAccessor da;
      :
  void setDbAccessor(DbAccessor obj) { da = obj; }
  void register(String who) { ... da.commit(who); ... }
  void cancel(String who) { ... da.commit(who); ... }
}
```

When the factory object creates an instance of Registration, it calls the setDbAccessor method to set the da field to an appropriate sub component. Note that the definition of Registration does not include the code for creating the sub component.

The dependency injection enables switching the sub component without editing the source code of the component for business logic. It allows the reuse of the components without the sub component. If the dependency injection is not used, the source code of the component would include the initialization code for creating the sub component for accessing a particular type of database. It would heavily depend on that particular type of database and thus it must be always reused with the sub component together. For example, the definition of Registration would be changed into the following if the dependency injection is not used:

```
class Registration {
  DbAccessor da;
      :
  Registration() { da = new MySQLAccessor(); }
  void register(String who) { ... da.commit(who); ... }
  void cancel(String who) { ... da.commit(who); ... }
}
```

The constructor must explicitly create a sub component. To change the type of the sub component, the definition of Registration must be edited.

Although the dependency injection addresses code dependency with respect to the initialization of the links between a component and its sub components, the code dependency is not limited to that. Another is the code dependency due to method calls. If a component calls a method on such a sub component as Logging, the code of that component depends on the code of the sub component since it includes an expression for the method call. To eliminate that expression for independently reusing the component without that sub component, the source code of that component must be modified. This type of dependency can be partly reduced by the pointcut-advice mechanism of AspectJ since the pointcut-advice mechanism enables *so-called* implicit method calls without including an expression for a method call in the code of the caller-side component. However, the reduction by AspectJ is limited.

To enable reducing dependency among components, GluonJ separates aspect bindings and aspect implementations as JBoss AOP [17], Caesar [26], and JAsCo [30] do. The aspect binding specifies which joinpoint an advice body is executed at. In AspectJ, it corresponds to pointcuts. The aspect implementation implements the main logic of the concern for that aspect. In AspectJ, both the aspect bindings and the aspect implementations are declared in an aspect. They are not separated at all. In GluonJ, the aspect bindings are described in XML and the aspect implementations are described as objects written in regular Java. We can thereby use the AOP technology for connecting normal Java objects.

Furthermore, GluonJ gives developers the full control of the instantiation of an aspect implementation. This is a unique feature of GluonJ. An aspect of AspectJ and an aspect implementation of JBoss AOP etc. are implicitly created on demand by the runtime systems of the languages. Developers can only select one of the instantiation schemes provided in advance. For example, the default scheme of AspectJ is singleton; only a single instance is created for each aspect implementation. When the thread of control reaches a joinpoint specified by a pointcut, the corresponding advice is executed on that single instance.

The Logging concern in GluonJ: The aspect binding of GluonJ includes glue code written in Java. It connects an instance of an aspect implementation to a target component. The following XML code is the aspect binding of GluonJ for Logging:

```
<aspect>
  <injection>
    Logger Clock.aspect = new Logger();
  </injection>
  <advice>
    <pointcut>
      execution(void Clock.paint(Graphics))
    </pointcut>
    <before>
      Logger.aspectOf(this).print();
    </before>
  </advice>
</aspect>
```

The pointcut selects as joinpoints method calls to the paint method in Clock. When the thread of control reaches these joinpoints, the code surrounded by the before is executed. Although this code block seems similar to AspectJ's before advice, it is not executed on an instance of Logging since it is glue code. Rather, it is executed on an instance of Clock as part of the paint method. The code block obtains a Logging object associated as an aspect to the Clock object and then calls the print method on the Logging object. The association of the Logging object is specified by the block surrounded by the injection tag. The definition of the Logging is a regular class definition as following:

```
public class Logging {
  private PrintStream output = System.out;
  public void setStream(PrintStream out) {
    output = out;
  }
  public void print(String m) {
    output.println(m);
  }
}
```

Note that the code blocks surrounded by injection and before are written in regular Java. This fact provides the full control of aspect binding for the developers. The injection block may associate any instance of Logging; it does not have to create a new instance of Logging for each Clock object. It even does not have to associate an aspect to a Clock object. It can assign an appropriate object to an existing field as the mechanism of the dependency injection performs. Also, the before block does not call a method on the associated aspect. It may call a method on the object pointed to by another field of Clock.

5 Summary

This tutorial first discussed that program translators will be able to simplify a framework protocol, which is often complicated in compensation for its functionality and reusability. Developing a program translator only for a particular framework is too expensive, but the reflection technology provides a powerful platform for such a program translator. If an appropriate implementation technique is used, the inefficiency of reflective computing can be reduced. This tutorial also mentions aspect-oriented programming (AOP). It can be also used for implementing a program translator and it provides a simpler programming model than the reflection technology. Although the reflection and AOP technologies significantly improved our ability for simplifying a framework protocol, the current states of those technologies are never perfect. For example, we need further study for dealing with the example in Section 2.1.

References

1. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A.: Common lisp object system specification. Sigplan Notices (1988) (X3J13 Document 88-002R).
2. Cazzola, W.: mcharm: Reflective middleware with a global view of communications. IEEE Distributed System On-Line **3** (2002)
3. Chiba, S.: A metaobject protocol for C++. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. Number 10 in SIGPLAN Notices vol. 30, ACM (1995) 285–299

4. Chiba, S.: Macro processing in object-oriented languages. In: Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98), IEEE Press (1998) 113–126
5. Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000) 313–336
6. Chiba, S., Ishikawa, R.: Aspect-oriented programming beyond dependency injection. In: ECOOP 2005. LNCS 3586, Springer-Verlag (2005) pp.121–143
7. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture. In: Proc. of the 7th European Conference on Object-Oriented Programming. LNCS 707, Springer-Verlag (1993) 482–501
8. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient Java bytecode translators. In: Proc. of Generative Programming and Component Engineering (GPCE '03). LNCS 2830, Springer-Verlag (2003) 364–376
9. Dahm, M.: Byte code engineering with the javaclass api. Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin (1999)
10. Ershov, A.: On the essence of compilation. In Neuhold, E., ed.: Formal Description of Programming Concepts, North-Holland (1978) 391–420
11. Ferber, J.: Computational reflection in class based object oriented languages. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1989) 317–326
12. Foote, B., Johnson, R.E.: Reflective facilities in Smalltalk-80. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1989) 327–335
13. Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html (2004)
14. Futamura, Y.: Partial computation of programs. In: Proc. of RIMS Symposia on Software Science and Engineering. Number 147 in LNCS (1982) 1–35
15. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley (1983)
16. Golm, M., Kleinöder, J.: Jumping to the meta level, behavioral reflection can be fast and flexible. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 22–39
17. JBoss Inc.: JBoss AOP 1.0.0 final. http://www.jboss.org/ (2004)
18. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. The MIT Press (1991)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP'97 – Object-Oriented Programming. LNCS 1241, Springer (1997) 220–242
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001 – Object-Oriented Programming. LNCS 2072, Springer (2001) 327–353
21. Maes, P.: Concepts and experiments in computational reflection. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1987) 147–155
22. Masuhara, H., Matsuoka, S., Asai, K., Yonezawa, A.: Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1995) 300–315
23. Masuhara, H., Matsuoka, S., Watanabe, T., Yonezawa, A.: Object-oriented concurrent reflective languages can be implemented efficiently. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1992) 127–144

24. Masuhara, H., Yonezawa, A.: Design and partial evaluation of meta-objects for a concurrent reflective languages. In: ECOOP'98 - Object Oriented Programming. LNCS 1445, Springer (1998) 418–439
25. McAffer, J.: Meta-level programming with coda. In: Proc. of the 9th European Conference on Object-Oriented Programming. LNCS 952, Springer-Verlag (1995) 190–214
26. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 90–99
27. Smith, B.C.: Reflection and semantics in Lisp. In: Proc. of ACM Symp. on Principles of Programming Languages. (1984) 23–35
28. Smith, B.: Reflection and semantics in a procedural languages. Technical Report MIT-TR-272, M.I.T. Laboratory for Computer Science (1982)
29. Sun Microsystems: Java 2 Platform, Enterprise Edition (J2EE). (http://java.sun.com/j2ee/)
30. Suvée, D., Vanderperren, W., Jonckers, V.: Jasco: An aspect-oriented approach tailored for component based software development. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 21–29
31. Tatsubori, M., Chiba, S., Killijian, M.O., Itano, K.: Openjava: A class-based macro system for java. In Cazzola, W., Stroud, R.J., Tisato, F., eds.: Reflection and Software Engineering. LNCS 1826, Springer Verlag (2000) 119–135
32. Watanabe, T., Yonezawa, A.: Reflection in an object-oriented concurrent language. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (1988) 306–315
33. Welch, I., Stroud, R.: From dalang to kava — the evolution of a reflective java extension. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 2–21