Josh: An Open AspectJ-like Language

Shigeru Chiba
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku, Tokyo 152-8552,
Japan
chiba@is.titech.ac.jp

Kiyoshi Nakagawa
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku, Tokyo 152-8552,
Japan
nakagawa@csg.is.titech.ac.jp

ABSTRACT

Although aspect-oriented programming (AOP) is becoming widely used, the design of the pointcut language and the generic and reusable description of advice are still research topics. To address these topics, this paper presents *Josh*, which is our new AspectJ-like language with an extensible pointcut language and a few mechanisms for generic description. The extensible pointcut language is based on the idea of open compiler. Since Josh allows defining a new pointcut designator in Java, the users can define a pointcut designator useful in a particular application domain. Also, Josh allows any Java expression to be included in the body of advice. This mechanism enables the generic and reusable description of advice.

Keywords

Pointcut, generic description, extensibility.

1. INTRODUCTION

Aspect-oriented programming (AOP) [13] is an emerging technique for modularizing *crosscutting* concerns, which cut across several basic modules. These concerns cannot be modularized with existing techniques such as object-oriented programming. There have been several AOP languages and systems [1, 20, 22] and AspectJ [15] is a typical AOP language for Java.

Although AspectJ is getting widely used for software development, a few challenging issues are known in the community. The first one is to extend the pointcut language, that is, the language for specifying pointcuts. The current one provides only limited capability to specify execution points. Another issue is generic description. The programmers must often repeatedly describe similar inter-type declaration (previously called *introduction*) for several classes but those descriptions should be replaced by a single generic description. Although enabling parameterized description [10] similar to the C++ templates might be a partial solution, the current version of AspectJ does not provide even such a mechanism.

This paper presents our efforts to address these issues. Our approach is to develop *Josh*, which is an AspectJ-like language with an extensible pointcut language and a few mechanisms for generic description. The extensible pointcut language is based on the idea of the open-compiler approach, which was first proposed by Lamping et al [17] and has been actively studied by several researchers [3, 11, 24, 21, 23]. This approach is not to make the source code of the compiler open to the public. It is rather to develop easily understandable abstraction of the internal structure or behavior of the compiler and to provide the programming interface to customize the compiler through that abstraction. This paper shows our abstract model of the AspectJ compiler and programming interface for language customization.

Josh allows expert developers to develop domain-specific extensions to the pointcut language in Java so that execution points can be selected with a complex algorithm. These extensions might be somewhat ad hoc but they can be reused by other application developers. In general, such domain-specific extensions should not be included in the language specifications but they should be supplied to the users as an optional library or a compiler plug-in. As for the issue of generic description, Josh can automatically adjust the description of the inter-type declaration to fit a particular class at compile time according to an algorithm given in Java. Although only experts would be able to write such an algorithm, other application developers can reuse it.

In Section 2, we first present two motivating examples of the extensions. In Section 3, we propose our AspectJ-like language named *Josh*. Section 4 shows an example of the extensions in Josh. Related work is discussed in Section 6. Section 7 concludes this paper.

2. MOTIVATIONS

First of all, we show motivating examples to illustrate limitations of the current version of AspectJ.

2.1 Pointcut language

The figure-editor example [15] is frequently used to explain the idea of AOP. In this example, figure elements are represented by several subclasses of FigureElement: Line, Rectangle, and so on (Figure 1). A crosscutting concern in this example is to update the window of the editor when the shape of a figure element is changed. The code for window updates spreads over all the classes of figure elements such as Line.

In AspectJ, this concern is modularized to be an aspect in the following way. First, the programmer defines a pointcut to select all the method calls, such as a call to setWidth in Rectangle for changing the shape of a figure element. Then she defines advice to execute a method call to repaint on the window object when the thread of control reaches one of these method calls. repaint successively calls redraw on every figure element so that the window will be updated. The pointcut will enumerate the signatures of all such methods as setWidth or it will include a pattern string that matches these signatures. For example, the pattern string like *.set*(..) matches all the calls to the methods the names of which begin with set.

In the pointcut language of AspectJ, the programmer must enumerate all the methods that change the shape of a figure element or follow the programming convention in which the names of such methods start with set. However, AspectJ does not provide any mechanism for ensuring that all the methods are correctly enumerated or named with that programming convention.

A better solution is to use a more sophisticated algorithm for selecting method calls. For example, the redraw method would read some fields of the FigureElement object. The methods we have to select are ones updating the values of those fields. It would be good if we could describe a pointcut that selects method calls according to this algorithm.

Unfortunately, AspectJ does not allow us to describe such an algorithmic pointcut. The description of pointcuts consists of several *pointcut designators*; they specify a kind of execution such as method calls and field access, or simple conditions that the selected execution points (called *join points* in AspectJ) must satisfy. Such conditions filter out execution points, for example, with a method signature, which may include wild cards, or the class that the execution points belong to. The conditions can be composed with a logical operator such as &&. In AspectJ, however, we cannot specify a condition that uses the dependency among classes as we showed with the example above. AspectJ in a future version may provide a new pointcut designator for supporting the pointcut we showed above but providing a large number of pointcut designators would make the language difficult to learn especially if most of them are used only for specific applications. In fact, the AspectJ designers seem to try to keep the language simple and elegant as much as possible.

2.2 Generic description

In AspectJ, an aspect can include a method or field declaration in another class. This is called inter-type declaration. Suppose that we are implementing a tree-traversal program according to the Visitor pattern [7] in AspectJ. The tree represents an arithmetic expression. If we use Java, the tree-traversal concern will be implemented with the Visitor class and the accept methods in all the tree-node classes (Figure 2). The accept method calls the visitXX method (XX is a node-class name) on the Visitor object given as the parameter to accept.

In AspectJ, this concern is implemented as an aspect. The definitions of the Visitor class and the accept methods are put in the aspect. However, the programmer must repeatedly write the definition of accept for every node class; if there are ten node classes, she must write the definitions of ten accept methods, which are only slightly different from each other. For example,

```
void Sum.accept(Visitor v) {
  v.visitSum(this);
}
void Number.accept(Visitor v) {
  v.visitNumber(this);
}
```

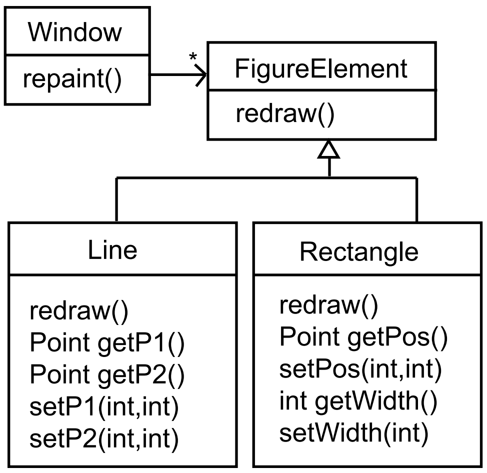

Figure 1: Figure Editor

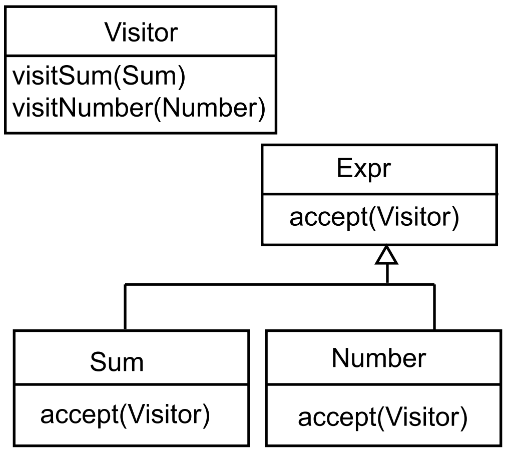

Figure 2: Arithmetic Expression

The differences between the two definitions are only the name of the method called on v and the class declaring accept.

To avoid this redundancy, it would be good if the accept methods could be defined in a more generic form without repetitions:

```
void Expr+.accept(Visitor v) {
  Class nodeClass = this.getClass();
  String name = "visit" + nodeClass.getName();
  Method m = v.getClass().getMethod(name,
                   new Class[] { nodeClass });
  m.invoke(v, new Object[] { this });
}
```

This inter-type declaration appends the accept methods to all the subclasses of Expr. No redundant repetitions are required. However, the body of accept must be described with the Java reflection API [12] and thus the execution performance of this method involves performance penalties. Furthermore, the description of the method body is complicated and difficult to read. It first obtains the class name of this object and concatenates "visit" and that class name. Then it obtains the Method object representing the method with that concatenated name. It next makes an array of Object containing the value of this as the parameter. Finally, it calls invoke on the Method object with v and that array.

Although there is no redundancy in this generic inter-type declaration, the readability is rather worse than that of the redundant description shown at the beginning. The problem would be that AspectJ provides only a limited number of mechanisms for generic description. For example, AspectJ provides special syntax proceed, which is available in the body of around advice. proceed executes the computation that is originally supposed to do at the execution point specified by the pointcut. It can be regarded as a mechanism for generic description since it is independent of details of that computation such as the method name and parameters and thus a single body of around advice may cover several execution points that have different details. The programmer does not have to repeatedly write a similar but different advice body for each execution point. However, proceed and any other syntax of AspectJ do not simplify the declaration of the accept methods shown above.

3. JOSH

To address the problems mentioned in the previous section, we propose a new AspectJ-like language named *Josh*, which allows the users to define a new pointcut designator suitable for their applications. While Josh currently supports a subset of the specifications of AspectJ, it provides several new language mechanisms for generic description.

3.1 Programming model of AspectJ

We first describe the programming model of AspectJ to present our terminology used in this paper. In AspectJ, a modular unit of crosscutting concern is an *aspect.* A component of the aspect is either *aspect member*, *inter-type declaration*, or *advice.* The aspect member is a field or method belonging to the aspect. The inter-type declaration was formerly called introduction. It typically adds a new field or method to another class.

The advice specifies a piece of code executed at some well-defined points of the program execution. It is a pair of *pointcut* and *body.* The body represents the executed code and the pointcut represents where the body should be executed. There are three kinds of advice: before, after, and around.

The inter-type declaration can be also regarded as a pair of pointcut and body although the specifications of AspectJ does not say so. For example,

```
int Point.getX() { return x; }
```

This appends the getX method to only the Point class. If Point+ is substituted for Point, the classes that the method is appended to are extended to all the subclasses of Point. Point and Point+ can be regarded as the pointcut of this inter-type declaration. The body is the rest of this declaration.

Pointcuts are classified into two categories: *static designators* and *dynamic designators.* The static designators depend on only the lexical information obtained by static program analysis. AspectJ's call and get are static designators. The dynamic designators depend on the information available only at runtime. AspectJ's cflow and target[1] are dynamic designators. The inter-type declaration cannot use these dynamic designators; it must use only the static designators since the target language is Java, in which class definitions cannot be altered during runtime.

Another element of the components of an aspect is context exposure. In AspectJ, we can define a pointcut parameter so that the runtime context at the execution point, such as method parameters, can be exposed and available in the body of the advice. We call this *context exposure.* Providing pointcut parameters for the inter-type declaration is also feasible and useful although AspectJ does not provide them. In fact, Josh provides a framework to access the lexical context of the program for the inter-type declaration.

3.2 The Design of Josh

Josh is our AspectJ-like language. Although Josh does not currently support all the functionality of AspectJ, it allows the users to extend the pointcut language. The syntax supported by Josh is a subset of AspectJ's. For example, the following program is a logging aspect written in Josh:

```
aspect Logging {
  before(): call("void Point.set*(..)") {
    System.out.println("Point was called");
  }
}
```

This prints a logging message if a method in Point is called and the name of that method starts with set. The syntax is the same as AspectJ's except the parameter to call is surrounded by double quotes. The reason why Josh uses this syntax is described in Section 3.4.

For inter-type declaration, however, Josh uses different syntax so that the pointcut is clearly separated from the body.

```
intro(): within("Point") {
  int getX() { return x; }
}
```

This appends the getX method to the Point class. The within designator specifies the class to which the method in the following block is appended. intro represents that the following block is the inter-type declaration. This design decision different from AspectJ is just for making the extension mechanism for inter-type declaration, which will be shown later, syntactically consistent with one for advice.

If the method is not preceded by intro, the method is regarded as an aspect member. For example,

```
static void print(String s) {
  System.out.println(s);
}
```

This declares the print method in the aspect. To call this method, the programmer must explicitly write Logging.print, where Logging is the aspect name. In the current implementation of Josh, the fully qualified name is required to access a field or method declared in an aspect. Finally,

[1]target is not a static designator since it depends on the runtime type of the target object.

```
intro(): within("Point"):
  implements("Comparable");
```

This changes the type hierarchy so that the class Point implements the Comparable interface. Note that there is : (colon) between within and implements. The designators following the colon are not part of the where specification but part of the body of the component.

Josh provides a mechanism for context exposure. Unlike AspectJ, Josh does not provide a designator like args for exposing runtime context but it provides special variables available in the body of the advice. These variables can be used without the explicit declaration of the use of the variables. This Perl-like syntax is provided by Javassist, which is the underlying system of Josh, for efficient execution and generic description [5].

An example of these variables is $0, which represents the target object if the execution point is a method call or a field access. $1, $2, ... represent the values of the first, second, ... parameters to the method. They correspond to args of AspectJ. $_ represents the result value. Assignment to $_ within the advice body changes the result value of the execution point. $1 and $2 are used as follows:

```
before(): call("void *.move(int,int)") {
  if ($1 < 0 || $2 < 0)
    System.err.println("assertion failure");
}
```

This before advice prints an error message before the call to move if the first or second parameter to move is negative. The types of $1 and $2 are determined according to the call designator.

A few special variables are used for reflective computation. $args represents an array of the parameters. Since the type of $args is java.lang.Object[], if some parameter types are primitive ones like int, they are implicitly wrapped by wrapper objects like Integer. $sig (signature) represents an array of java.lang.Class representing the formal parameter types. $type represents the java.lang.Class object representing the formal return type. $class represents the java.lang.Class object representing the type of the target object.

In the body of around advice, another special syntax $proceed is available. It corresponds to AspectJ's proceed. The around advice traps the execution at the specified point and the advice body is run instead of that execution point. The original action associated with that execution point must be invoked with $proceed if it is needed. For example,

```
around(): call("void *.move(int,int)") {
  if ($1 < 0 || $2 < 0)
    throw new Exception();
  else
    $_ = $proceed($1, $2);
}
```

This advice runs the original call to move with the original parameters unless the parameters are negative. $_ represents the result value of this execution point. Josh also provides special syntax $$, which can be used as the parameter to $proceed. It represents the list of all the parameters $1, $2, ... Thus $proceed($1, $2) above can be replaced with $proceed($$) without changes of the behavior. $$ is unique syntax of Josh; it can encapsulate the number of the parameters and thus, for example, it helps to write a reusable advice body that can be used for the calls to methods with different signatures. For example, the users can simply write:

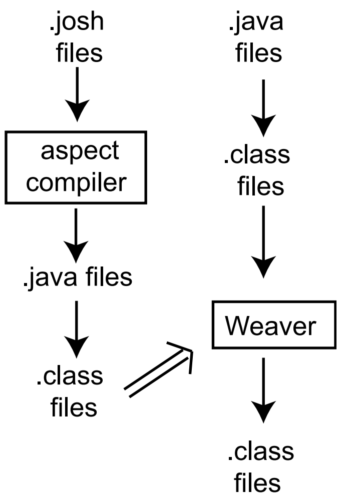

Figure 3: The Josh Compiler

```
$_ = $proceed($$);
```

to execute the original action of the execution point trapped by around, whatever the kind of that execution point is. The users can use the above statement as a convenient idiom.

Some users might dislike the Perl-like syntax. Josh allows those users to use not $1, $2, ... but parameter names appearing in the source code if the pointcut is execute.[2] Extending Josh to provide the args designator of AspectJ is our future work so that variable names given by the users can be bound to $1, $2, ... We believe that this extension would be quite straightforward.

3.3 Josh Compiler

The Josh compiler consists of an aspect compiler and a weaver. The aspect compiler is a source-to-source translator from the Josh language to the Java language. It reads the aspect definitions written in Josh (.josh files) and translates them into Java programs (.java file). According to these compiled aspect definitions, the weaver transforms the regular Java classes compiled by a regular Java compiler such as javac. The output of this weaver is the compiled Java program (.class files) in which the aspects have been woven with the classes. Figure 3 illustrates this flow. Note that the aspect compiler or the weaver do not require souce code of the regular Java classes. They only require .class files.

We here present how aspect definitions (.josh files) are translated into regular Java programs. Understanding the overview of this translation is required for the users to extend the Josh compiler.

For every component of an aspect, the weaver must insert a piece of code at appropriate positions in the class definitions. The positions are specified by the lexical part of the pointcut, or *the shadow* [19], in the component. If the component is an aspect member or an inter-type declaration, then the piece of code is inserted as a field or method declaration. If it is advice, then the piece of code is inserted in a method body. The dynamic designators of the pointcut, if any, are contained in the inserted code so that the

[2]The source code must be compiled with the -g option so that the symbol table is included in the .class file.

Table 1: Part of the methods in CtClass

String	getName() get the class name
void	setName(String name) change the class name
int	getModifiers() get the class modifiers such as public
void	setModifiers(int m) change the class modifiers.
CtClass	getSuperclass() get the super class
void	setSuperclass(CtClass c) change the super class
CtClass[]	getInterfaces() get the interfaces
void	setInterfaces(CtClass[] i) change the interfaces
CtField[]	getFields() get all the fields
void	addField(CtField f) add a new field
CtMethod[]	getMethods() get all the methods
void	addMethod(CtMethod m) add a new method
CtConstructor[]	getConstructors() get all the constructors
void	addConstructor(...) add a new constructor

advice body is executed only if the requirements given by the dynamic designators are satisfied.

To do this code insertion, the aspect compiler translates aspect definitions into a Java program using our compile-time reflection library called *Javassist* [4]. Javassist provides the functions of reading a class file (compiled binary) and creating several objects representing the class, the fields, the constructors, and the methods contained in the class file. These are CtClass, CtField, CtConstructor, and CtMethod objects.[3] We call them join-point objects.[4]

These objects provide methods for inspecting the class definition with respect to the static structure (Table 1 and 2). These methods are parallel to ones defined in the standard reflection API [12]. Since Javassist is for manipulating a class that has not been loaded yet, it does not support object creation, method invocation, or field access. Unlike the reflection API, however, Javassist enables to alter the class definition. The API design of Javassist is based on the structural reflection, which was first developed in Smalltalk-80 [8]. We later discuss the Javassist API again.

The weaver traverses the join-point objects included in every given .class file and, if it finds the position where a piece of code must be inserted, then the weaver inserts the code there through Javassist. The weaver inquires of the compiled aspect definitions whether it inserts the code or not. The aspect compiler translates every component of an aspect into an if statement, which is executed for inquiry as part of the weaver program. For example, if an aspect

[3]Ct means compile-time. It is added to distinguish from the standard classes Class, Field, and so on.

[4]Although a join point means a dynamic execution point in AspectJ, a join-point object in Josh represents a lexical point of the program.

Table 2: Part of the methods in CtMethod

String	getName() get the method name
void	setName(String name) change the method name
int	getModifiers() get the method modifiers such as public
void	setModifiers(int m) change the method modifiers
void	setBody(String src) change the method body
void	instrument(ExprEditor e) modify the method body
void	insertBefore(String src) insert the code at the beginning of the body
void	insertAfter(String src, boolean asFinally) insert the code at the end of the body

Table 3: Part of the methods in MethodCall

CtClass	getCtClass() get the class of the target object
CtMethod	getMethod() get the callee method
CtMethod	getMethodName() get the name of the callee method
CtClass[]	mayThrow() get the exceptions that may be thrown
CtBehavior	where() get the method body containing this call
int	getLineNumber() get the line number of the source line
String	getFileName() get the source file
void	replace() replace the method-call expression

includes an inter-type declaration for adding a new field z to the Point class, that declaration is compiled into the following if statement:

```
if (c.getName().equals("Point"))
  c.addField(CtField.make("int z;", c));
```

Here, c is the CtClass object representing the given class. The method including this if statement is loaded together with the weaver program and it is invoked for the CtClass object created from every given class file at weaving time. Note that the lexical part of the pointcut is translated into the conditional expression in the if statement.

Javassist also produces an object representing an expression contained in a method body. Such an object is either MethodCall, NewExpr (object creation by the new operator), FieldAccess, Handler (exception handler), Cast, or InstanceOf object. They are also join-point objects and provide methods for inspecting the static context of the expressions (Table 3). To obtain these join-point objects, the instrument method must be called on a CtMethod object. The parameter to the instrument method is an ExprEditor object, which is an event listener; whenever the instrument method finds a join point in a method body, it calls the edit method on the ExprEditor object. The parameter to the edit method is the object representing that join point, for example, a MethodCall object. The following is an example of the use of the instrument method (suppose that mth is a variable referring to a CtMethod object).

```
mth.instrument(new ExprEditor() {
  public void edit(MethodCall expr) {
    // do something here.
  });
```

The edit method is invoked whenever a method-call expression is found in the method body represented by the CtMethod object.

Advice might be translated into an if statement accessing these join-point objects. For example, if the advice is:

```
around(): call("void *.move(..)") {
  System.out.println("move");
  $_ = $proceed($$);
}
```

then the aspect compiler produces something like this:

```
new ExprEditor() {
  public void edit(MethodCall mc) {
    if (mc.getMethodName().equals("move")) {
      mc.replace(
        "{ System.out.println(\"move\");"
      + "$_ = $proceed($$); }");
    }
  }}
```

This if statement is run for every MethodCall object found in a method body. Here, mc is a MethodCall object representing the method-call expression at the caller side. The replace method compiles the given code, which is an advice body, into the Java bytecode and substitutes the bytecode for the original one. Thus, the advice body is inlined at the join point. replace also statically expands the special variables starting with $. The runtime values accessed through those special variables are saved to local variables in the prologue of the compiled advice body. The special variables are compiled to the bytecode that accesses these local variables.

If the pointcut is composition of two pointcut designators, then the translated designators are also composed with the same logical operator. For example, This pointcut:

```
call("void *.move(..)") && within("Display")
```

is translated into the following conditional expression:

```
mc.getMethodName().equals("move")
  && c.getName().equals("Display")
```

Here c is the CtClass object representing the class that includes a method body containing the method-call expression represented by mc. The if statement including this expression is run if the MethodCall object mc is found.

If the if statement requires multiple join-point objects, it is run when the weaver finds the innermost object among them. The join-point objects representing an expression contained in a method body are the inner objects of the CtMethod object representing the method with that body. The CtMethod object is an inner object of the CtClass object representing the class declaring that method. The if statement requiring multiple join-point objects is run with the innermost object and the outer objects surrounding that innermost object. For example, if the innermost object is a MethodCall object, then the if statement also receives the CtMethod and CtClass objects surrounding that MethodCall object. If no unique innermost object can be determined, a compile error is reported. For example, the following pointcut:

```
call("void *.move(..)") && get("int Point.x")
```

is translated into an if statement requiring MethodCall and FieldAccess objects. Since either of the two is not the inner object of the other, this pointcut causes a compile error. In fact, the pointcut above does not select any join points.

Note that only the lexical part of a pointcut, or *the shadow* [19], is translated into a conditional expression. The rest of the pointcut, which depends on runtime context, is translated into part of the inserted code. For example, this around advice:

```
around(): call("void *.move(..)")
    && within("Display") && target("Point") {
  System.out.println("point");
  $_ = $proceed($$);
}
```

is translated into this Java code:

```
if (mc.getMethodName("move")
          && c.getName().equals("Display")) {
  mc.replace(
    "if ($0 instanceof Point){"
    + " System.out.println(\"point\");"
    + "$_ = $proceed($$); }");
}
```

Here, c is the CtClass object and mc is the MethodCall object. The inserted code examines at runtime whether the target object is an instance of Point or not. Note that the code shown above is naive; for better runtime performance, the target type should be statically checked as well as at runtime.

The declarations of fields and methods belonging to an aspect are processed differently from other components of the aspect. If there are such declarations in an aspect, the aspect compiler produces the definition of a class with the same name as the aspect. Then the aspect compiler includes all those declarations in that class as static fields or methods. The current implementation of Josh supports only singleton aspects. It does not support per-object aspects.

3.4 Extensibility

Josh allows the users to define a new pointcut designator for both inter-type declaration and advice. The definition of a new designator is given as a static method written in Java with using Javassist. Such a static method can implement a complex algorithm for identifying join points since it can fully use the ability of Javassist for inspecting the static structure of the programs.

All occurrences of a new designator appearing in pointcuts are translated by the aspect compiler of Josh into calls to the static method implementing the new designator. The calls are put in the conditional expressions of the if statements corresponding to the pointcuts. The called static method must return a boolean value.

Suppose that we want to define a simple designator named paramType1. This captures a method the first parameter of which is the given class type. It is used as follows:

```
paramType1("ColorPoint")
```

This picks out a method if the type of the first parameter is ColorPoint. The definition of this designator is the following static method in Java (this is a simplified version that ignores error recovery):

```
static boolean paramType1(CtMethod m,
        String[] args, JoshContext jc) {
  CtClass parType
   = m.getMethod().getParameterTypes()[0];
  CtClass argType = jc.getType(args[0]);
  if (parType.subtypeOf(argType))
    return true;

  if (argType.subtypeOf(parType)) {
    jc.setIf("$1 instanceof " + argType.getName());
    return true;
  }
  else
    return false;
}
```

The first parameter to paramType1 is a join-point object that this designator examines. It can be any other join-point object like CtClass or MethodCall. The second parameter is an array of String, which represents parameters given to the designator. The user-defined designators can receive any number of parameters, which are comma-separated multiple String literals surrounded by double quotes. The third parameter is a JoshContext object containing house-keeping information. The return value is true if the first parameter of the method represented by the given CtMethod object matches the type given to the designator.

Since the first parameter is a CtMethod object, the Josh weaver executes the if statement including the call to paramType1 whenever the weaver finds a CtMethod join-point object (and it is the innermost object). The paramType1 returns true if the first parameter type might be the specified type, for the example above, ColorPoint. The weaver inserts the advice body only if the paramType1 method returns true.

Dynamic designator and context exposure

The paramType1 designator is dynamic, that is, it needs runtime type check. The weaver must also insert an if statement to surround the inserted advice body so that the body will be executed only if the result of the runtime type check is true. Note that Josh deals with two if statements: one is executed at weaving time for *the shadow* [19] of pointcuts and the other is at runtime for the rest.

To insert an if statement executed at runtime, the paramType1 method invokes setIf in JoshContext. The parameter to setIf is a conditional expression that will be included in the if statement for the runtime type check. If two designators composed with a logical operator give different conditional expressions to the setIf method, these expressions are also composed with the same logical operator.

The JoshContext class provides the expose method to define a new variable for exposing execution context. If the static method implementing a designator calls this method, the given source text is included in the code block inserted as the advice body. Hence, if a variable is declared in the source text passed to expose, that variable is available within the advice body. For example, the following call makes a variable cname available in the advice body. The value of cname is the name of the class associated with the execution point.

```
jc.expose("String cname = \"" + c.getName()
          + "\";");
```

Here, jc is a JoshContext object and c is the CtClass object. The given source text is compiled and inserted at the beginning of the advice body. If the class represented by c is Point, then the inserted code block is:

```
{ String cname = "Point";
  /* advice body */ }
```

Although this example exposes lexical context, the expose method can be used as well for exposing dynamic context with a variable name specified by the user. Any value can be exposed if constructing the source text for obtaining the value is possible.

The JoshContext class also provides methods for accessing outer join-point objects surrounding the innermost one. Recall that the method implementing a new designator only receives the innermost join-point object as a parameter.

Inter-type declaration

Josh also allows the users to define a new designator used as the body of the inter-type declaration. Recall that the inter-type declaration in Josh uses different syntax from in AspectJ, for example,

```
intro(): within("Point"):
  implements("Comparable");
```

The body of this declaration is implements, which makes the Point class implement the Comparable interface. Josh allows defining a new designator that can be used instead of implements.

To implement it, the user defines a static method in Java. Like other user-defined designators, the method receives a join-point object, an array of String, and the JoshContext object as parameters. The return type is void. The type of the join-point object must be equal to the type of the join-point examined by the pointcut. If not, the Josh compiler reports an error. We below show an example of the method:

```
static boolean addThrows(CtMethod m,
        String[] args, JoshContext jc) {
  CtClass type = jc.getType(args[0]);
  m.addExceptionType(type);
}
```

This appends the given exception type to the throws list of the method captured by the where specification.

Body

For generic and reusable description of aspects, Josh allows any Java expression to be contained in the body of inter-type declaration or advice. This mechanism can be used to modularize the concern of the Visitor pattern mentioned in Section 2.2:

```
intro(): within("Expr+") {
  void accept(Visitor v) {
    v.visit<% josh.getCtClass().getName() %>(this);
  }
}
```

This inter-type declaration appends the accept method to every subclass of Expr. The code fragment surrounded by <%

and %> can be any expression in Java. The expression must evaluate to a String object. The resulting text is substituted for the code between <% and %> at compile time.

In the expression, a special variable josh is available. It is a reference to the current JoshContext object. For the example above, the name of the class specified by the pointcut is obtained from josh. The aspect compiler of Josh translates the body of the inter-type declaration into the following String concatenation, which constructs the code inserted by the weaver:

```
"void accept(Visitor v) { v.visit"
+ josh.getCtClass().getName()
+ "(this);}"
```

Note that the appended methods differ among the subclasses since the code between <% and %> is re-evaluated at compile time for every subclass.

This inter-type declaration is simple and efficient compared to ones in Section 2.2. Since the declaration in Josh exploited compile-time reflection [3], it does not involve runtime penalties unlike the one in AspectJ.

3.5 Javassist

Since all the extensions to Josh are implemented with the compile-time reflection library called Javassist [4], the range of the extensibility and the easiness of the extension depend on the ability of Javassist. For example, the Josh users cannot define a new kind of join point; only the fixed set of join points provided by Javassist is available. The position where an advice body can be inserted is either before, after, or around. To overcome these limitations, Javassist itself must be extended.

The current version of Javassist provides the ability to inspect and alter the type signatures of the given class, such as a field type and a method signature. The ability to inspect them is equivalent to that of the Java reflection API. Adding a new field or method to a class is also possible. The definition can be given as source text.

The ability to inspect and alter part of a method body is currently limited. As for this ability, the CtMethod object allows inserting a code fragment at the beginning or end of the method body. A catch clause can be added to the method for receiving an exception thrown in the method body. The CtMethod also provides a mechanism for obtaining the set of join-point objects included in the method body; the set may include throw and instanceof expressions, which AspectJ cannot handle. Control or data flow information among join points is not available. However, the join-point objects provide various kinds of information. For example, the MethodCall object provides the name and signature of the called method, exceptions that the method-call expression may throw, the method containing the expression, and so on. Furthermore, the join-point objects provide the replace method, which substitutes the given source text for the original expression in the method body.

Despite the limited ability of Javassist, a relatively complex designator such as AspectJ's cflow can be implemented within the confines of Javassist. cflow can be implemented with a thread-local variable, which is incremented (or decremented) when a method starts (or finishes). Javassist can insert the code performing this at necessary places.

4. EXAMPLE

Since the motivating example shown in Section 2.2 has been implemented by Josh in Section 3.4, this section mentions how to address the problem shown in Section 2.1. We present a user-defined designator updater for selecting the method calls specified by the algorithm based on the dependency among classes. The updater designator is used as follows:

```
updater("FigureElement", "redraw")
```

This designator can be implemented on top of Josh since Javassist, the backend of Josh, provides the necessary information.

The following static method is the implementation of the updater designator:

```
static boolean updater(MethodCall mc,
        String[] args, JoshContext jc) {
  CtClass root = jc.getCtClass(args[0]);
  String mname = args[1];
  CtMethod mth = mc.getMethod();

  // skip if the method is redraw().
  if (mth.getName().equals(mname))
    return false;

  Hashtable fields
    = enumerateFields(jc, root, mname);
  updated = false;
  mth.instrument(new ExprEditor() {
    public void edit(FieldAccess expr) {
      String name = expr.getFieldName();
      if (expr.isWriter()
          && fields.get(name) == expr.getCtClass())
        updated = true;
    }
  });
  return updated;
}
```

This method returns false if the called method mth is redraw. Otherwise, it invokes enumerateFields for enumerating the fields that the redraw methods in a subclass of FigureElement read. These fields are recorded in the hashtable fields (for better performance, the value of fields should be cached). Then, the updater method calls instrument method to examine whether the body of the called method mth contains the assignment to the field recorded in the hash table. updated is a static field of the class declaring the updater method. If the body includes such assingment, the updater method returns true.

The instrument method enumerates join points included in the body. Whenever it finds a FieldAccess join-point, it calls the edit method on the given ExprEditor object. The parameter to edit is that FieldAccess object. ExprEditor is a class provided by Javassist.

The enumerateFields method examines the redraw methods and records all the fields that are declared in a subclass of FigureElement and also read by one of the redraw methods. The fields found are recorded in the hash table. Suppose that m is the CtMethod object representing the redraw method and root is the CtClass object representing the FigureElement class. The following code examines the method and records the fields:

```
m.instrument(new ExprEditor() {
  public void edit(FieldAccess expr) {
    if (expr.isReader()
        && expr.getCtClass().subclassOf(root))
      fields.put(expr.getFieldName(),
                 expr.getCtClass());
  }
});
```

Here, fields is a HashTable object.

5. EXPERIMENT

Although Josh is still under development, we show the results of simple benchmark tests. We defined before advice that counts up the number of non-static method calls on the same object. This advice was woven at all the method calls (caller-side) included in benchmark programs, which are taken from the sequential benchmarks of the Java Grande forum[5].

Table 4 lists the results. *Original* represents the execution time of the benchmark programs without the before advice. *Josh* and *AspectJ* represent the execution time of the programs in which the before advice is woven by Josh or AspectJ. The results show that the runtime performance of Josh is comparable with that of AspectJ. Since Josh saves all the method-call parameters to local variables in the prologue of the compiled advice body, the overhead of Josh is larger if the program includes methods that take a number of parameters.

6. RELATED WORK

The if pointcut designator of AspectJ allows any Java expression to be included in a pointcut. The advice body is executed only if the expression is true at runtime. This designator could be regarded as an extension mechanism of the pointcut language. However, the if designator enables us to implement only the dynamic designators. Since the static designators evaluate at compile time and thus imply no runtime overheads, new designators should be implemented as static ones as long as they are independent of runtime context. Josh supports both dynamic and static user-defined designators.

Lieberherr et al proposed statically executable advice [18]. This allows the compile-time execution of an advice body, for example, to check if a programming rule is enforced. Josh can be used to implement this statically executable advice.

There are a few extensible AOP languages. For example, some researchers [2, 9] proposed to use a logic language for describing a complex pointcut. In this language, pointcut designators are logic predicates. Logic reasoning is definitely useful to select a set of join points satisfying complex conditions but the users must learn logic programming. We took the opposite direction for Josh. Since Josh is an AOP language for Java, the language for extending Josh is also Java, which the users should be familiar with. Using Java would reduce initial learning costs to extend Josh.

Kiczales proposed new pointcut designators such as pcflow for AspectJ to address the problem mentioned in Section 2.1 [14]. Developing a general-purpose designator is a right approach although we believe there are always problems that the general-purpose designator cannot cover.

There have been several open compilers. Their differences are the functions and internal data structures exposed to the users. Intrigue [17] is an open Scheme compiler; it allows customization of code generation. MPC++ [11] and OpenJava [24] open up a parse tree so that it can be transformed. They also enable syntax extension. Josh opens up not a parse tree but part of the weaving process represented with the join-point objects.

Developing the exposed structure such as the join-point objects is not trivial. If the AspectJ compiler is naively made open, it might force the users to deal with low-level bytecode image as typical toolkits for transforming Java class files, such as BCEL [6] and JMangler [16], do. Letting the users define a new pointcut designator for that system would not be practical. On the other hand, the join-point objects of Josh provide source-level abstraction while keeping practical expressive power.

[5]`http://www.epcc.ed.ac.uk/javagrande`

7. CONCLUDING REMARKS

This paper mentioned why aspect-oriented languages such as AspectJ should be extensible and then proposed an open-compiler solution for making those languages extensible. Our AspectJ-like language called *Josh* allows the users to implement a new pointcut designator in Java. This mechanism enables selection of join points without complicated programming conventions or error-prone description of the pointcut. Josh also allows a Java expression to be included within an inter-type declaration. This mechanism enables avoiding redundant description of the inter-type declaration.

Although AspectJ provides only a fixed set of built-in pointcut designators, it allows the users to compose pointcut designators with logical operators to define a new pointcut. A problem is that the built-in designators are too high-level and hence some pointcuts like one in Section 2.1 cannot be defined in AspectJ. Our idea was to provide lower-level join-point objects as primitives and use a regular language like Java for composing them to define new pointcuts. This approach provides better flexibility. The built-in designators of AspectJ can be implemented on this platform as user-defined desingators for convenience.

Since the join-point objects represent the structural facet of the program, Josh does not support the definition of a pointcut depending on other facets such as data flow. Extending Josh to cover other facets is our future work. Furthermore, Josh has not supported all the features of AspectJ, such as type checking and named pointcuts. Extending Josh to support all of them is also our future work.

8. REFERENCES

[1] Aksit, M., L. Bergmans, and S. Vural, "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach," in *ECOOP '92*, LNCS 615, pp. 372–395, Springer-Verlag, 1992.

[2] Brichau, J., K. Mens, and K. D. Volder, "Building Composable Aspect-specifc Languages with Logic Metaprogramming," in *Generative Programming and Component Engineering (GPCE 2002)* (D. Batory, C. Consel, and W. Taha, eds.), LNCS 2487, pp. 93–109, Springer, 2002.

[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.

Table 4: The elapsed time (sec.)

	Euler	Molecular	Monte Carlo	Ray Tracer	Search
Original	28.0	22.1	22.7	19.1	17.4
Josh	28.5	22.1	23.7	24.4	19.4
AspectJ	28.2	22.1	23.1	22.1	20.2
# of calls	2,307	7,753	310,045	5,338,398	71,228,058

Sun Blade 1000 (Dual UltraSPARC III 750MHz, 1GB memory), Solaris 8, Sun JDK 1.4.0_01, AspectJ 1.1b2.

[4] Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.

[5] Chiba, S. and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," in *Proc. of Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pp. 364–376, Springer-Verlag, 2003.

[6] Dahm, M., "Byte Code Engineering with the JavaClass API," Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.

[7] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison-Wesley, 1994.

[8] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[9] Gybels, K. and J. Brichau, "Arranging Language Features for More Robust Pattern-based Crosscuts," in *Proc. of 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD 2003)*, pp. 60–69, ACM Press, 2003.

[10] Hanenberg, S. and R. Unland, "Parametric Introductions," in *Proc. of 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD 2003)*, pp. 80–89, ACM Press, 2003.

[11] Ishikawa, Y., A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota, "Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —," in *Proc. of Reflection 96*, pp. 153–166, Apr. 1996.

[12] Java Soft, "Java^{TM} Core Reflection API and Specification." Sun Microsystems, Inc., 1997.

[13] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pp. 220–242, Springer, 1997.

[14] Kiczales, G., "The Fun Has Just Begun." Keynote talk at 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD 2003), 2003.

[15] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp. 327–353, Springer, 2001.

[16] Kniesel, G., P. Costanza, and M. Austermann, "JMangler — A Framework for Load-Time Transformation of Java Class Files," in *Proc. of IEEE Workshop on Source Code Analysis and Manipulation*, 2001.

[17] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, "An Architecture for an Open Compiler," in *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.

[18] Lieberherr, K., D. H. Lorenz, and P. Wu, "A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ," in *Proc. of 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD 2003)*, pp. 40–49, ACM Press, 2003.

[19] Masuhara, H., G. Kiczales, and C. Dutchyn, "Compilation Semantics of Aspect-Oriented Programs," in *Proc. of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2002, pp. 17–26, 2002.

[20] Mezini, M. and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 97–116, 1998.

[21] Ogawa, H., K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura, "OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java," in *ECOOP 2000*, LNCS 1850, pp. 362–387, Springer-Verlag, 2000.

[22] Ossher, H. and P. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pp. 734–737, 2000.

[23] Tanter, E., N. Bouraqadi, and J. Noyé, "Reflex – Towards an open reflective extension of Java," in *Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pp. 25–43, Springer, 2001.

[24] 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, pp. 119–135, Springer Verlag, 2000.