Pipa: A Behavioral Interface Specification Language for AspectJ*

Jianjun Zhao[1] and Martin Rinard[2]

[1] Department of Computer Science, Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

[2] Laboratory for Computer Science, Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139, USA
rinard@cag.lcs.mit.edu

Abstract. Pipa is a behavioral interface specification language (BISL) tailored to AspectJ, an aspect-oriented programming language. Pipa is a simple and practical extension to the Java Modeling Language (JML), a BISL for Java. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. This paper discusses the goals and overall approach of Pipa. It also provides several examples of Pipa specifications and discusses how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification. The goal is to facilitate the use of existing JML-based tools to verify AspectJ programs.

1 Introduction

Aspect-oriented programming (AOP) has been proposed as a technique for improving the separation of concerns in software design and implementation [2,13, 17,19]. AOP provides explicit mechanisms for capturing the structure of crosscutting aspects of the computation such as exception handling, synchronization, performance optimizations, and resource sharing. Because these aspects crosscut the dominant problem decomposition, they are usually difficult to express cleanly using standard languages and structuring techniques. AOP can eliminate the code tangling often associated with the use of such standard languages and techniques, making the program easier to develop, maintain, and evolve.

The field of AOP has, so far, focused primarily on problem analysis, language design, and implementation. The specification and validation of aspect-oriented programs has received comparatively little attention.

To formally verify aspect-oriented programs, we must have some means to formally specify the properties of aspect-oriented programs. Note that, because

* This work was carried out during Jianjun Zhao's visit to Laboratory for Computer Science, Massachusetts Institute of Technology.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 150–165, 2003.

AOP introduces new concepts such as join points, advice, instruction, and aspects, existing formal specification languages can not be directly applied to AOP languages. This motivates us to design a formal specification language that is appropriate for specifying programs written in AOP languages.

Instead of designing a generic specification language for AOP, we choose instead to design a *behavioral interface specification language* (BISL) tailored to AspectJ, an aspect-oriented extension to Java [14]. A *behavioral interface specification* describes both the details of a module's interface with clients and its behavior from the client's point of view [16]. By using a BISL, we are able to formally specify both the behavior and the exact interface of AspectJ programs' modules, which is an essential step towards the formal verification of these modules.

Our BISL for AspectJ is called Pipa, which is a simple and practical extension to Java Modeling Language (JML) [16], a widely accepted BISL for Java. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. Pipa provides annotations to specify AspectJ programs with pre- and postconditions, class invariants, and aspect invariants. These annotations enable both dynamic analysis in support of activities such as debugging and testing and static analysis in support of the formal verification of properties of AspectJ programs. Static analysis activities could verify that the code of advice of an aspect correctly implements its specification, the specification of advice in an aspect is compatible with the specification of the method in a class that the advice advises, and the correctness of the aspect weaving process.

The key to the verification process is to develop a transformation tool that automatically transforms an AspectJ program, together with its Pipa specification, into a corresponding Java program and JML specification. By doing so, some JML-based checking and verification tools [16,6,10] can be used directly to check and verify AspectJ programs.

In this paper, we discuss the goals of Pipa and its overall specification approach. We also provide examples of how to use Pipa to specify AspectJ aspects, and discuss how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification, which is a crucial step towards the utilization of existing JML-based tools to verify AspectJ programs.

The rest of the paper is organized as follows. Section+2 presents the design rationale for Pipa. Section 3 briefly introduces the Java Modeling Language. Section 4 uses some examples to show how AspectJ aspects are specified in Pipa. Section 5 discusses the issues about specification inheritance and crosscutting. Section 6 discusses how to transform an AspectJ program together with its Pipa specification into a standard Java program and JML specification. Section 7 discusses related work; we conclude in Sect. 8.

2 Design Rationale

Our purpose is to understand how to formally specify and verify aspect-oriented programs. Several questions focus our investigation. First, what is an aspect invariant, and how would we specify it? How would one specify aspects that contain around advice which may alter the return value of a method in the base code? What is the contract checking semantics between an aspect and the base code? How would one verify that the behavior of an aspect does not violate the desired functionality of the base code? How would one verify that aspects whose advice affects the behavior of base code still provide an acceptable semantics according to the specification of the base code? How would one verify the correctness of a woven program after weaving the aspect and base code?

Designing Pipa is therefore only a part of our proposed activities. We must also develop techniques and tools to support the formal specification and verification of AspectJ programs augmented with Pipa specifications. We have therefore chosen to design Pipa as a compatible extension to JML to (1) facilitate its adoption by current JML users, and (2) facilitate the adoption of existing JML-based tools to check AspectJ programs. JML is an especially appropriate base for the Pipa design for two reasons. First, AspectJ is a seamless extension to Java for implementing crosscutting concerns, and JML is a BISL specially designed for Java. By structuring Pipa as an extension to JML, we can focus our attention on the new issues associated with the use of aspects. Second, (JML has efficient tool support for both static and dynamic checking of Java programs) if we can transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification, we can use JML-based tools directly to verify AspectJ programs. Based on these considerations, we keep the following issues in mind to make Pipa as compatible with JML as possible.

- Each legal JML specification for Java should also be a legal Pipa specification for AspectJ.
- Specifying AspectJ programs with Pipa should feel like a natural extension of specifying Java programs with JML.
- It should be possible to extend existing JML-based tools (such as static checking tools, run-time assertion checking tools, documentation tools, and design tools) to support Pipa in a natural way.
- The Pipa specifications themselves should be strongly connected to the AspectJ code, as JML specifications are connected to Java code.

Like JML, Pipa specifications are also expressed as Javadoc-style comments in AspectJ interface definitions, enclosed between `/**` and `*/`. Pipa therefore permits the specification to be embedded in regular AspectJ files.

To focus on the key ideas of Pipa, in this paper we do not consider the specification of AspectJ classes and interfaces. These classes and interfaces can be specified in Pipa in a similar way as JML [16]. Moreover, we only consider join points related to method and constructor calls, and introductions that introduce members such as methods and constructors.

3 The Java Modeling Language

The Java Modeling Language (JML) [16] is a formal BISL tailored to the Java programming language [7]. JML allows assertions (pre- and postconditions and class invariants) to be specified for Java classes and interfaces. JML adopted the model-based approach of Larch [8] by supporting specification-only model fields. These fields describe abstractly the value of objects and are used only for specification purposes. The predicates in JML are written using regular Java expressions extended with logical operators and universal and existential quantifiers.

JML specifications are expressed as special comments in Java interface definitions, following `//@` or enclosed between `/*@` and `*/`. JML also permits the specification to be embedded in regular Java files. In addition, JML specifications can also be expressed as Javadoc-style comments, that is, enclosed between `/**` and `*/`.

JML supports specifying a class at both the method and the class level. These two specifications together form the complete behavioral specification for the class. For example, the following shows a simple method-level specification:

```
/**@  public normal_behavior
   @    requires x >= MIN_X && x <= MAX_X;
   @    ensures true;   */
public void moveX(int x) { ...  }
```

The specification states that if a precondition (represented by a `requires` clause) `x >= MIN_X && x <= MAX_X` holds at the call of a public method, the method terminates normally, i.e., does not throw an exception, and the postcondition (represented by an `ensures` clause) `true` is satisfied at the end of the method call. There is a variation for the `normal_behavior` specification, i.e., a `behavior` specification, which can be used to specify the conditions under which a method may, may not, or must throw an exception.

JML also provides class-level specifications with additional clauses such as `invariant`, `constraint`, and `model`. A `model` clause allows the declaration of so-called model variables which are variables that exist only within a specification. Such variables are often used to refer to the internal state of an object. An `invariant` clause in JML declares those properties that are true in all publicly visible, reachable states of an object, i.e., for each state that is outside of a public method's execution. An invariant is supposed to be established by the class constructors and to be preserved by each (public) method. A `constraint` clause is used to specify how values may change between earlier and later states, such as a method's pre-state and its post-state.

In addition, a specification in JML may be composed of several cases separated by the keyword `also`; which states that when the precondition of one case holds, the rest of that case's specification must be satisfied. JML also supports specification inheritance. A subtype inherits the specifications from its supertype's public and protected members (i.e., fields and methods), as well as its invariants and history constraints as additional specification cases.

4 Aspect Specifications

In AspectJ, an aspect is a modular unit for implementing crosscutting concerns. Its definition is similar to a Java class, and can contain methods, fields, and initializers. Pointcuts and advice support the implementation of crosscutting aspects. Pipa can specify an aspect at both the module and the aspect level. Pipa uses *module-level specifications* to specify the behavior of individual modules such as advice, introduction, and methods in an aspect, and *aspect-level specifications* to specify the global properties of the aspect as a whole. In this section, we show how advice and introduction can be specified using Pipa; method specification in Pipa is the same as in JML. Through this paper, we introduce the specification approach of Pipa with the use of an example AspectJ program (taken from [1] with slight modifications), which consists of two classes `Point` and `Line` (both shown in Fig. 1) and several aspects.

4.1 Advice Specifications

Advice defines pieces of code in an aspect that should be executed when a pointcut is reached during the execution of a program. AspectJ provides three kinds of advice, that is, *before*, *after*, and *around* advice [1]. The most significant feature of advice is that it can dynamically change the behavior of a class it advises, and therefore implements *behavioral crosscutting.*

In Pipa, the specification of a piece of advice is similar to that of a method in JML: the advice is annotated with preconditions, postconditions, and frame conditions; these are declared, respectively, with `requires`, `ensures`, and `modifies`

```
class Point {
  int x, y;
  /**@  model instance int x_Mdl, y_Mdl; */

  /**@  depends x_Mdl <- x; */
  /**@  represents x_Mdl <- x; */

  /**@  depends y_Mdl <- y; */
  /**@  represents y_Mdl <- y; */

  /**@  public behavior
     @    assignable x_Mdl;
     @    ensures x_Mdl == x;
     @    signals (Exception z) false; */
  public void setX(int x) {
     this.x = x;
  }
  /**@  public behavior
     @    assignable y_Mdl;
     @    ensures y_Mdl == y;
     @    signals (Exception z) false; */
  public void setY(int y) {
     this.y = y;
  }
}
```

```
class Line {
  private Point p1, p2;
  /**@  model instance Point p1_Mdl, p2_Mdl; */

  /**@  private depends p1_Mdl <- p1; */
  /**@  private represents p1_Mdl <- p1; */

  /**@  private depends p2_Mdl <- p2; */
  /**@  private represents p2_Mdl <- p2; */

  /**@  public behavior
     @    assignable p1_Mdl;
     @    ensures p1_Mdl == p1;
     @    signals (Exception z) false; */
  public void setP1(Point p1) {
     this.p1 = p1;
  }
  /**@  public behavior
     @    assignable p2_Mdl;
     @    ensures p2_Mdl == p2;
     @    signals (Exception z) false; */
  public void setP2(Point p2) {
     this.p2 = p2;
  }
}
```

Fig. 1. Two classes `Point` and `Line` with their Pipa specifications

```
aspect PointBoundsPreCondition {
 /**@  public behavior
    @    requires x >= MIN_X && x <= MAX_X;
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires x < MIN_X || x > MAX_X;
    @    ensures false;
    @    signals (Exception z)
    @      z instanceof RuntimeException; */
 before(int x): call(void Point.setX(int))
                 && args(x) {
     if ( x < MIN_X || x > MAX_X )
        throw new RuntimeException();
 }
}
```

(a)

```
aspect PointBoundsPostCondition {
 /**@  public behavior
    @    requires p.getX() == x;
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires p.getX() != x;
    @    ensures false;
    @    signals (Exception z)
    @      z instanceof RuntimeException; */
 after(Point p, int x): call(void Point.setX(int))
                       && target(p) && args(x) {
    if ( p.getX() != x )
       throw new RuntimeException();
 }
}
```

(b)

Fig. 2. (a) A piece of before advice with its Pipa specification. (b) A piece of after advice with its Pipa specification

clauses. These preconditions, postconditions, and frame conditions together form the *specification* of the advice, which can be used to verify the code of the advice.

In contrast to the pre- and postcondition of a method, which are associated with method call and return, the pre- and postcondition of advice is defined in terms of the principle of control flow transferring [5]. This is because each piece of advice in AspectJ is automatically woven into the method(s) it advises by a compiler (called `ajc`) during the aspect weaving process. Advice therefore executes in response to certain program actions, instead of being directly invoked like a method. From this viewpoint, the pre- and postcondition of a piece of advice can be defined as follows:

- A *precondition* for a piece of advice is an assertion that states the properties that must hold before the control flow is transferred into the advice.
- A *postcondition* for a piece of advice is an assertion that states the properties that the advice must establish before the control flow returns to the advised method.

Before Advice. Before advice executes when a join point is reached and before the computation proceeds [1]. It may execute when computation reaches the method call and before the actual method starts executing. In Pipa, the behavior of before advice can be specified by a precondition, a postcondition, and a frame condition.

As an example, consider the aspect `PointBoundsPreCondition` shown in Fig. 2a, which declares a piece of before advice that modifies the behavior of the class `Point`'s `setX` method. The advice can be applied to each join point where a target object of type `Point` receives a method call with signature `void Point.setX(int)`. The `args` keyword denotes the argument of the method call.

The specification of this before advice is associated with the advice code, which contains two specification cases combined by a keyword `also`. For the first case, a `requires` clause supplies a normal precondition, which must be satisfied before control transfers to the advice from the method `setX`, and an `ensures` clause provides a normal postcondition, which must hold before control transfers to the advised method `setX`. The specification states that if the advice is entered with `x >= MIN_X` and `x <= MAX_X`, control must flow to the advised method `setX`. The implicit frame condition for this case means that no relevant locations may be assigned when this precondition holds. The second specification case states that if the advice is entered with `x < MIN_X` or `x > MAX_X`, the control must return to the caller of the method `setX` by throwing a `RuntimeException`.

After Advice. After advice executes after the computation under the join point terminates [1]. It may execute after the method body has executed, and just before control is returned to the caller of the method. AspectJ supports three kinds of after advice, that is, *after*, *after-returning*, and *after-throwing* advice. After-returning advice executes just after each join point, but only when the advised method returns normally. After-throwing advice executes just after each join point, but only when the advised method throws an exception of type `Exception`. after advice executes just after each join point, regardless of whether the advised method returns normally or throws an exception. Like before advice specifications, after advice specifications have a precondition, a postcondition, and a frame condition.

As an example, consider the aspect `PointBoundsPostCondition` shown in Fig. 2b, which declares a piece of normal after advice that modifies the behavior of class `Point`'s `setX` method. The after advice can be applied to each join point where a target object of type `Point` receives a method call signature `void Point.setX(int)`. The `target` and `args` keywords denote the target object and the argument of the method call.

The Pipa specification for the after advice has two specification cases. The first case states that if the advice is entered with `p.getX() == x`, the control must flow to the advised method. The second case states that if the advice is entered with `p.getX() != x`, control must return to the caller of the method `setX` by throwing a `RuntimeException`.

Around Advice. Around advice executes when a join point is reached, and has explicit control over whether the computation under the join point is allowed to execute at all [1]. around advice differs from before advice and after advice in the sense that it may execute code both before and after the method, and by (optionally) executing the `proceed` call, which causes the original method under the join point to execute. To specify around advice, Pipa borrows some ideas from [5]. Pipa uses the `proceeds` *predicate* clause, taken from [5], to state that when control flow proceeds to the original method body (or to any additional advice if present), the around advice must make *predicate* hold. Pipa also uses a keyword `then`, also taken from [5], to divide the specification of the around

```
aspect PointBoundsEnforcement {
 /**@  public behavior
    @    requires x >= MIN_X && x <= MAX_X;
    @    proceeds true;
    @    signals (Exception z) false;
    @  then
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires x < MIN_X;
    @    proceeds false
    @    signals (Exception z) false;
    @  then
    @    assignable p.x_Mdl;
    @    ensures p.x_Mdl == MIN_X;
    @    signals (Exception z) false
    @ also
    @  public behavior
    @    requires x > MAX_X;
    @    proceeds false;
    @    signals (Exception z) false;
    @  then
    @    assignable p.x_Mdl;
    @    ensures p.x_Mdl == MAX_X;
    @    signals (Exception z) false  */
  void around(Point p, int x):
       call(void Point.setX(int))
              && target(p) && args(x) {
    if (x >= MIN_X && x <= MAX_X ) {
       proceed(p, x);
    } else if (x < MIN_X) {
       p.setX(MIN_X);
    } else {
       p.setX(MAX_X);
    }
  }
```

Fig. 3. A piece of around advice with its Pipa specification

advice into two parts: the *before part*, which is the portion of the specification corresponding to the around advice before proceeding to the original method, and the *after part*, which is the portion corresponding to the around advice after returning from the original method but before returning to the original caller. With these specification constructs, Pipa can specify around advice conveniently.

As an example, consider the aspect `PointBoundsEnforcement` shown in Fig. 3, which declares a piece of around advice to modify the behavior of class `Point`'s `setX` method. The around advice can be applied to each join point where a target object of type `Point` receives a call to its method with signature `void Point.setX(int)`. The `target` and `args` keywords are used to assign the name to the target object and the argument of the method call. The code of around advice states that if `x` is greater than `MIN_X` and less than `MAX_X`, then the statement `proceed(p, x);` causes control flow to transfer to the original `setX` method body with the same arguments as the original invocation. Otherwise, in the `else` clauses the advice calls the `setX` method on `Point` directly.

The specification of the around advice, which is associated with the advice code, has three specification cases combined using the keyword `also`. The first case applies when `x >= MIN_X` and `x <= MAX_X`. The `proceeds true` clause states that control flow can always proceed to the original method (`setX`) body. The part following the `then` keyword states that after control flow transfers to the original method (`setX`) body, it then returns to the original client.

The second case, following the first `also` keyword, concerns the case where `x < MIN_X`. The `proceeds false` clause states that the control never proceeds to the original method (`setX`) body. The part following the `then` keyword also has a `requires` clause as a postcondition of the case. The `assignable` and `ensures` clauses state that control may return to the original client with possible mutation to `p`'s `x_Mdl` model field and with the given postcondition predicate satisfied. The third case, following the second also keyword, which concerns the case where x > MAX_X, can be explained similarly.

4.2 Introduction Specifications

While a piece of advice can change the behavior of the classes it crosscuts, it can not change the static type structure of the classes. AspectJ provides a form called *introduction* that can operate over the static structure of type hierarchies [1]. An aspect can use introduction to add new fields, constructors, or methods into given classes or interfaces.

Pipa's introduction specification mechanism is similar to its method specification mechanism. Each piece of introduction may be annotated with preconditions (declared by requires), postconditions (declared by ensures), and frame conditions (declared by modifies). These preconditions, postconditions and frame conditions together form the *specification* of the introduction, which can be used to verify the code of the introduction.

As an example, consider the aspect Introduction shown in Fig. 4, which uses the new keyword to introduce a constructor into the Point and Line classes. These two constructors have two int parameters and the same body. The Pipa specification for the introduction is tailored to the code and states that if the introduction is called with x >= 0, the call must return to the caller normally.

4.3 Aspect Invariants

The previously discussed pre- and postconditions specify properties of individual modules such as advice, introduction, and aspect methods. There is also a need, however, to express global semantic or integrity properties for the aspect as a whole. Aspect invariants express these kinds of properties. Such invariants may involve only attributes, attributes and modules, or different modules in an aspect. Informally, an invariant of an aspect is a set of assertions (i.e., invariant clauses) that each instance of the aspect will satisfy at all times when the state is observable. Pipa uses an invariant clause, borrowed from JML, to specify aspect invariants.

As an example, consider the Buffering aspect shown in Fig. 5, which implements a buffering function. The first aspect invariant states that the value of the

```
aspect Introduction {
   /**@ public behavior
      @   assignable x_Mdl;
      @   requires x >= 0;
      @   ensures x_Mdl == x;
      @   signal (Exception z) false;   */
   public int (Point || Line).new(int x) {
        this.x = x;
   }
}
```

Fig. 4. A piece of introduction with its Pipa specification

```
public aspect Buffering pertarget(target(FileOutputStream)) {
  /**@ public invariant counter >= 0;
     @ public invariant buff =! null && buff.length == BUFF_SIZE; */
  private static final int BUFF_SIZE = 256;
  int counter = 0;
  byte[] buff = new byte[BUFF_SIZE];

  pointcut writeByte(byte[] bytes):
  void around(byte[] bytes): throws IOException: writeBytes(bytes) { }
}
```

Fig. 5. An aspect `Buffering` with aspect invariants

aspect's field `counter` must be greater than or equal to zero. The second states that the aspect's field `buff` is not null and the array it refers to has exactly `BUFF_SIZE` (256) elements.

5 Aspect Specification Inheritance and Crosscutting

We next discuss aspect specification inheritance and crosscutting in Pipa.

5.1 Specification Inheritance

The inheritance rules in AspectJ are (1) an aspect can only extend an abstract aspect; it can not extend a concrete aspect, (2) an aspect can extend a class, and (3) an aspect can implement any number of interfaces. According to these inheritance rules, we design some specification inheritance rules for Pipa as follows.

- A subaspect inherits the specifications of its superaspect's public and protected members (fields, methods, advice, introductions, and pointcuts), as well as the public and protected aspect invariants.
- A subaspect inherits the specifications of its superclass's public and protected members (fields and methods), as well as the public and protected class invariants.
- A subaspect inherits the specifications of its superinterface's public and protected members (fields and methods), as well as the public and protected interface invariants.

These aspect inheritances can be thought of as textually copying the public and protected specifications of the advice, introduction, or methods of an aspect's superaspects and superclasses and all interfaces that an aspect implements into the aspect's specification and combining the specifications using `also` keyword. By the semantics of advice, introduction, or method combining using `also`, in addition to any explicitly specified behaviors, these behaviors must all be satisfied by the advice, introduction, or method.

5.2 Specification Crosscutting

AspectJ supports both structural crosscutting (by means of introduction) and behavioral crosscutting (by means of advice) to modify the type structure and the behavior of classes an aspect crosscuts. Pipa should also be able to specify these structural and behavioral crosscutting issues at specification level. To make these possible, in the following we design some specification crosscutting rules for Pipa.

- The specification of an aspect's advice crosscuts the specifications of those classes' methods that the advice crosscuts (*behavioral crosscutting*).
- The specification of an aspect's introduction crosscuts the specifications of those classes that the introduction crosscuts (*structural crosscutting*).

These crosscutting rules ensure that an aspect specifies the structural and behavioral crosscutting of the one or more classes it crosscuts. This crosscutting can be thought of as syntactically (1) weaving the specification of a piece of advice in an aspect into the specification of each advised method in a class or different classes, and (2) weaving the specification of a piece of introduction in an aspect into the specification of one or more classes augmented by the introduction.

As an example of behavioral crosscutting, consider the aspect `DisplayUpdating` shown in Fig. 6, which modifies the behavior of methods in classes `Point` and `Line` shown in Fig. 1. `DisplayUpdating` declares a piece of after advice that can be applied to each join point where a target object of type `Point` receives a call to the method with signature either `void Point.setX(int)` or `void Point.setY(int)`. Also this after advice can be applied to each join point where a target object of type `Line` receives a call to the method with either signature `void Line.setP1(Point)` or signature `void Line.setP2(Point)`. In this case, the after advice in `DisplayUpdating` can affect the behavior of these methods in `Point` and `Line`. The specification of the after advice therefore should crosscut the classes `Point` and `Line`. This crosscutting can be thought of as syntactically weaving the specification of the after advice in `DisplayUpdating` into the specification of each advised method `setX`, `setY`, `setP1`, or `setP2`.

As an example of structural crosscutting, consider the aspect `Introduction` shown in Fig. 4, which publicly introduces two methods, one in class `Point`

```
aspect DisplayUpdating {
  pointcut move():
    call(void Line.setP1(Point)) ||  call(void Line.setP2(Point)) ||
    call(void Point.setX(int))   ||  call(void Point.setY(int))

  after(): move() {
    Display.update();
  }
}
```

Fig. 6. A piece of after advice that crosscuts two classes `Point` and `Line`

and another in class Line. According to the specification crosscutting, the Pipa specification for the introduction should be woven into the specification of both class Point and class Line.

6 Transforming Pipa Specifications Back to JML

One important reason to design Pipa based on JML is that we hope to make use of existing JML-based tools. To make this possible, we propose to develop a tool to automatically transform an AspectJ program together with its Pipa specification into a standard Java program and JML specification. To this end, we propose to modify the AspectJ compiler (ajc) to retain the comments associated with advice and aspect introduction during the weaving process. ajc supports Javadoc style and can retain comments of classes and interfaces during the weaving process. But this process does not retain comments of advice and introduction. By modifying the ajc, we can make it retain the comments of advice and introduction as well, with the resulting woven program a standard Java program with JML specifications. Therefore, after the transformation, all JML-based tools can be applied to AspectJ programs. However, when performing such transformations, we must find a way to handle certain problems such as those listed below.

The first problem is how to handle aspect invariants. A Pipa specification of a piece of advice or introduction can be directly transformed to a JML specification using a modified AspectJ weaver. However, the role of an aspect invariant in the weaving process is less clear, since aspect invariants may crosscut many different classes and should hold for all join points relevant to the advice. One conservative solution to this problem is to weave the aspect invariant into the class invariant of every class that the aspect crosscuts.

The second problem is how to handle the problem of specification weaving when weaving the aspects into classes. Several cases that are related to specification weaving must be considered because different advice may lead to different weaving rules. For example, a piece of after-returning advice, which is only valid for the case when the advised method returns normally, should not be woven into the specification of the advised method when the method returns by throwing an exception. Moreover, around advice that contains a proceed() construct also requires similar handling. However, for before advice or normal after advice, one can simply weave the specification into each method it advises with no additional adjustment. A transformation tool must correctly handle these different cases during the weaving process.

7 Related Work

Clifton and Leavens' [5] work on modular aspect-oriented reasoning also requires the specification of aspect advice. We see our work as differing from theirs in several ways. First, Pipa has a different goal. While the purpose of the Clifton and Leavens' work is to support modular aspect-oriented reasoning by extending

AspectJ with new language constructs, Pipa is intended to allow the full specification of AspectJ programs. Second, Pipa can specify the global properties of an aspect using aspect invariants that are different from pre- and postconditions for individual modules in the aspect. Third, Pipa supports aspect specification inheritance, and more importantly, Pipa supports aspect specification crosscutting, either structurally or behaviorally. Clifton and Leavens [5] focus on specifying advice in an aspect, not on issues about how to specify introduction, aspect invariant, aspect specification inheritance and crosscutting.

There has been significant work in the field of generic specification languages in general and BISLs in particular. Widely used generic specification languages include Z [21], VDM [11], and Larch [8]. Several BISLs that are based on Larch have been designed, each tailored to a specific programming language. Examples include LCL (for C) [8], LM3 (for Modula-3) [8], and Larch/C++ [4].

In addition to the Larch family, Meyer's work on the programming language Eiffel has advanced the cause of applying formal methods to object-oriented programs [18]. In Eiffel, unlike a Larch-style interface specification language, one can use Boolean expressions to specify pre- and postconditions for operations on abstract data types written in Eiffel, that is, program expressions can be used in pre- and postconditions. In addition, in Eiffel one can use class invariants to specify the global properties of instances of the class. On the other hand, several projects have been carried out to support the principle of Design By Contract (DBC), originally introduced by Meyer in Eiffel [18]. Examples include iContract [15] and Jass [3].

Recently, the emergence of Java as a popular object-oriented programming language has led to several BISLs designed for Java. Examples include JML [16], ESC/Java [6], and AAL [12]. JML allows assertions to be specified for Java classes and interfaces, and provide very expressive power to specify Java modules (classes and interfaces). ESC/Java is a static checking tool for Java. It can statically check for various errors in a Java program without executing the program. The annotation language in ESC/Java is a subset of JML that can be used to annotate Java code in various ways. AAL is an annotation language designed for annotating and checking Java programs. Like JML, AAL supports run-time assertion checking. AAL also supports full static checking for Java programs similar to ESC/Java. AAL translates annotated Java programs into Alloy [9], a simple first-order logic with relational operators, and uses Alloy's SAT solver-based automatic analysis technique to check Java programs. LOOP [10] is a project dedicated to verify JavaCard programs. LOOP adopts JML as its BISL for annotating Java modules and transforms annotated Java programs into a theorem-prover, PVS [20], to formally verify JavaCard programs.

Although the languages mentioned above can be used to specify programs written in various programming languages, they are not designed to specify programs written in AOP languages such as AspectJ. In summary, Pipa is the first BISL tailored to AspectJ that can be used to specify AspectJ programs. Pipa is also unique in its use of aspect invariants to specify the whole properties of an aspect, and in its supporting of aspect specification inheritance and crosscutting.

8 Concluding Remarks

In this paper we presented Pipa, a BISL tailored to AspectJ and discussed the goals of Pipa and the overall specification approach. Pipa is a simple and practical extension to JML. Pipa uses extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. We present several examples of Pipa specifications, and discuss how an AspectJ program together with its Pipa specification can be transformed into a corresponding Java program and JML specification, which is a crucial step towards the utilization of existing JML-based tools to verify AspectJ programs.

As future work, we would like to augment Pipa to support more kinds of join points, for example joint points at field accesses and dynamic join points such as `cflow` and `cflowbelow`. We also would like to refine our specification framework for AspectJ and to implement a tool to automatically transform an AspectJ program with Pipa specification into a corresponding Java program and JML specification.

Acknowledgements. We are grateful to Darko Marinov, Chandrasekhar Boyapati, and anonymous referees for valuable comments on an earlier version of the paper. We also thank Mik Kersten for valuable discussions on AspectJ implementation.

References

1. The AspectJ Team. The AspectJ Programming Guide. 2001. AspectJ home page: `http://www.aspectj.org`.
2. L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.
3. D. Bertetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Rosu, editors, *ENTCS*, Vol. 55, Elsevier Publishing, 2001.
4. Y. Cheon and G. T. Leavens. A Quick Overview of Larch/C++. *Journal of Object-Oriented Programming*, Vol.7, No.6, pp.39-49, October 1994.
5. C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. Technical Report TR#02-04, Department of Computer Science, Iowa State University, March 2002.
6. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp.234-245, June 2002.
7. J. Gosling, B. Joy, and G. Steele. The Java Language Specification. The Java Series, Addison-Wesley, Reading, MA, 1996.
8. J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A Modet, and J. M. Wing. Larch: Languages and Tools for Formal Specification. Springer-Verlag, New York, N. Y., 1993.
9. D. Jackson. Alloy: A Lightweight Object Modeling Notation. *ACM Transaction on Software Engineering and Methodology*, Vol.11, No.2, pp.256-290, April 2002.

10. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning About Java Classes (Preliminary Report). *Proc. ACM SIGPLAN 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp.329-340, October 1998.
11. C. B. Jones. Systematic Software Development Using VDM. Prentice-Hall, Englewood Cliffs, N.J., second edition, 1990.
12. S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language. *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2002.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proc. 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold. An Overview of AspectJ. *Proc. 15th European Conference on Object-Oriented Programming*, pp.327-352, LNCS, Vol.2072, Springer-Verlag, June 2001.
15. R. Kramer. iContract- the Java Design by Contract Tool. *Proc. Technology of Object-Oriented Language and Systems (TOOLS-USA)*, 1998.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report TR98-06, Department of Computer Science, Iowa State University, 1998 (Last version: June 2002).
17. K. Lieberher, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.
18. B. Meyer. Object-Oriented Software Construction. Prentice Hall, New York, N.Y., Second Edition, 1997.
19. H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2001.
20. S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, Vol.21, No.2, pp.107-125, February 1995.
21. J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, New York, N.J., Second edition, 1992.

Appendix: AspectJ

AspectJ [14] is a seamless aspect-oriented extension to Java by adding some new concepts and associated constructs to Java. These concepts and associated constructs are called join point, pointcut, advice, introduction, and aspect. We briefly introduce them in the following.

Aspect is modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect is defined by aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain state and methods, and also may be specialized in its sub-aspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. An aspect can *introduce* methods, attributes, and interface implementation declarations into types by using the *introduction* construct. Introduced members may be made visible to all classes and aspects (public introduction) or only within the aspect (private introduction), allowing one to avoid name conflicts by using pre-existing members. In addition to introduction, the essential mechanism provided for composing an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. An aspect can specify *advice* that is used to define some code that should be executed when a pointcut is reached. Advice is a method-like mechanism which consists of code that is executed *before*, *after*, or *around* a pointcut. around advice executes *in place* of the indicated pointcut, allowing a method to be replaced. As a class, an aspect can also be declared as abstract, that means it can not be instantiated. By default, a concrete aspect has only one instance exists for the program execution. Also named pointcuts can be declared abstract within an abstract aspect, allowing them to be given concrete definitions within concrete sub-aspects, much as abstract methods are used.

An AspectJ program can be divided into two parts: *base code* part which includes classes, interfaces, and other language constructs, and *aspect code* part which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that the base and aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].