Observers and Assistants:

A Proposal for Modular Aspect-Oriented Reasoning

by

Curtis Clifton and Gary T. Leavens

TR #02-04a
March 2002, Revised April 2002

Keywords: Observers, assistants, aspect-oriented programming languages, modular reasoning, specification, composition, AspectJ language, Java language, JML language, MultiJava language.

2002 CR Categories: D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects, control structures, inheritance, modules, packages, procedures, functions, and subroutines, advice, observers, assistants, aspects; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages, multiparadigm languages, AspectJ, JML; D.1.5 [*Programming Techniques*] Object-oriented programming — aspect-oriented programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics.

Copyright © Curtis Clifton and Gary T. Leavens, 2002. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning

Curtis Clifton and Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
+1 515 294 1580
{cclifton,leavens}@cs.iastate.edu

ABSTRACT

In general, aspect-oriented programs require a whole-program analysis to understand the semantics of a single method invocation. This property can make reasoning difficult, impeding maintenance efforts, contrary to a stated goal of aspect-oriented programming. We propose some simple modifications to AspectJ that permit modular reasoning. This eliminates the need for whole-program analysis and makes code easier to understand and maintain.

1. INTRODUCTION

Much of the work on aspect-oriented programming languages makes reference to the work of Parnas [23]. That work argues that the modules into which a system is decomposed should be chosen to provide benefits in three areas. Parnas writes (p. 1054):

> "The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood."

While much has been written about aspect-oriented programming as it relates to Parnas's second point, his third point is the primary concern of this paper. We contend that current aspect-oriented programming languages do not provide this third benefit in general, because they require systems to be studied in their entirety.

After describing and motivating the problem in this introduction, in Section 2 we propose a simple set of restrictions that, we believe, would bring these languages much closer to Parnas's ideal without any practical loss of expressiveness. This proposal is preliminary work and is presented in the hopes of generating discussion and feedback.

We begin by defining a notion of modular reasoning corresponding to Parnas's third benefit. Subsequent subsections in this introduction show how such modular reasoning is possible in the Java Programming Language™ [1, 9] but problematic in the current version of AspectJ™ [11].

For concreteness, our examples are shown in AspectJ. To support abstract reasoning we specify the examples using new aspect-oriented extensions to the Java Modeling Language (JML) [13, 14]. We believe our ideas are independent of Java and JML. We also believe that they are independent of the details of AspectJ and are generally applicable to the class of aspect-oriented languages.

1.1 Modular Reasoning

Before delving into the details, it is useful to define our terms. *Modular reasoning* is the process of understanding a system one module at a time. A language *supports modular reasoning* if the actions of a module *M* written in that language can be understood based solely on the code contained in *M* along with the specifications of any modules referred to by *M*. For example, in Java a single compilation unit, typically a file declaring a single top-level type (class or interface), is a module. The specification of that module is the behavior of objects of that type. Code is one form of specification. In a more expressive language, such as Eiffel [19] or Java annotated with JML, a specification can be given explicitly using pre- and postconditions, frame axioms, and invariants; such specifications serve as contracts that allow one to separately reason about the behavior and correctness of an implementation.

Our interest in modular reasoning in aspect-oriented programming languages is motivated in part by our earlier work on MultiJava [5, 6]. In that work we were concerned with modular static typechecking and compilation. This is closely related to the issue of modular reasoning, because the source code of a method body is a very precise behavioral specification of that method. A language that supports modular reasoning can therefore also permit separate compilation, as well as modular implementations of other tools (e.g., optimizers, verifiers, and model checkers). Thus, mechanisms that permit modular reasoning would have many benefits.

1.2 Modular Reasoning in Java

Java without aspect-oriented extensions supports modular reasoning. We illustrate this after giving some background on JML.

1.2.1. JML Background

Consider the examples in Figure 1 and Figure 2, modified slightly from Kiczales, *et al.* [11] and annotated with JML specifications. Specification annotations are enclosed in special comments; at-signs (`@`) at the beginning of lines in annotations are ignored.

In JML, *model fields*, like `xCtr` and `yCtr` in Figure 1, specify the abstract state of an object. They are specification-only constructs, but are treated formally as locations. The keyword `instance` says that they are considered to be model fields in all classes that implement the interface. A represents-clause (with keyword `represents`, as in Figure 2) says how the values of model fields are related to the actual, concrete fields of an object; and a depends-clause (also in Figure 2) allows such concrete fields to be assigned to when the model fields that depend on them are assignable [15].

In our JML examples, each method's specification is written preceding its signature. We use a desugared form of JML method specifications in this paper, in which a visibility level, which describes who can see the specification, is followed by the keyword `behavior`, which introduces a *specification case*. A specification case consists of several clauses. The forall-clause introduces logical

```
package foal02;

interface FigureElement {

   /*@ model instance int xCtr, yCtr; @*/

   /*@ public behavior
     @  forall int oldx, oldy;
     @  requires oldx == xCtr && oldy == yCtr
     @         && dx >= 0 && dy >= 0;
     @  assignable xCtr, yCtr;
     @  ensures xCtr == oldx + dx
     @      && yCtr == oldy + dy
     @      && \result == this;
     @  signals (Exception z) false; @*/
   FigureElement moveNE( int dx, int dy );

   /*@ public behavior
     @   ensures \result == xCtr;
     @   signals (Exception z) false; @*/
   /*@ pure @*/ int getX();

   /*@ public behavior
     @   ensures \result == yCtr;
     @   signals (Exception z) false; @*/
   /*@ pure @*/ int getY();
}
```

Figure 1: A Java module declaring an interface, with (unsugared) JML specifications

```
package foal02;

class Point implements FigureElement {

   private int _x = 0, _y = 0;

   /*@ private depends xCtr <- _x;
     @ private represents xCtr <- _x; @*/

   /*@ private depends yCtr <- _y;
     @ private represents yCtr <- _y; @*/

   /*@ public behavior
     @   assignable xCtr, yCtr;
     @   ensures xCtr == x && yCtr == y;
     @   signals (Exception z) false; @*/
   public Point( int x, int y ) {
      _x = x;  _y = y;
   }

   public /*@ pure @*/ int getX() { return _x; }

   public /*@ pure @*/ int getY() { return _y; }

   /*@ public behavior
     @   assignable xCtr;
     @   ensures xCtr == x;
     @   signals (Exception z) false; @*/
   public FigureElement setX( int x ) {
      _x = x;
   }

   /*@ public behavior
     @   assignable yCtr;
     @   ensures yCtr == y;
     @   signals (Exception z) false; @*/
   public FigureElement setY( int y ) {
      _y = y;
   }

   /*@ also
     @  public behavior
     @   requires dx < 0 || dy < 0;
     @   ensures false;
     @   signals (Exception z)
     @     z instanceof IllegalArgException;
     @*/
   public FigureElement moveNE( int dx, int dy ) {
      if (dx < 0 || dy < 0) {
         throw new IllegalArgException();
      }
      setX( getX() + dx );
      setY( getY() + dy );
   }
}
```

Figure 2: A Java module declaring a class

variables that are universally quantified over the specification case. The requires-clause gives the case's precondition, the assignable-clause its frame, the ensures-clause its normal postcondition, and the signals-clause its exceptional postcondition. Postconditions may use the keyword `\result` to refer to the value a method returns. Consider a call to the method being specified; for all assignments to the universally quantified variables that make the precondition true, the call may only mutate locations described by the frame, and if the call returns normally the method must satisfy the normal postcondition; if the call throws an exception it must satisfy the exceptional postcondition. For brevity we will omit empty forall-clauses, requires-clauses with the default predicate `true`, and assignable-clauses for which the frame has the default value of `\nothing`.

The form of method specifications we use in this paper is not the one users normally write in JML. But it is useful for our semantic study because it corresponds directly to the semantics. For example, JML borrows from Eiffel the ability to refer to the pre-state value of an expression *E* in a postcondition by writing `\old(`*E*`)`. The desugaring we assume here is to bind the pre-state values of each variable referred to in a `\old` expression to a fresh logical variable introduced in a forall-clause. JML also permits the use of calls to "pure" (side-effect free) methods in specifications, but in this paper we do not consider such calls. Instead we assume that calls to such methods are interpreted using the logical formulas in their normal postconditions. For example, one would normally write the specification of `moveNE` in Figure 1 as follows.

```
/*@ public behavior
  @  requires dx >= 0 && dy >= 0;
  @  assignable xCtr, yCtr;
  @  ensures getX() == \old(getX() + dx)
  @      && getY() == \old(getY() + dy)
  @      && \result == this;
  @  signals (Exception z) false;*/
FigureElement moveNE( int dx, int dy );
```

1.2.2. Java Examples

Suppose one wanted to write code that manipulates objects of type `FigureElement`. One could reason about such objects based solely on the information contained in Figure 1. That is, one would know the objects support a method named `moveNE` that takes two arguments of type `int` and that both arguments must be non-negative. Also if this precondition is satisfied, then the method will leave the object in a state where the values returned by `getX` and `getY` were increased by `dx` and `dy`, respectively.

Similarly, suppose one wanted to write code that manipulated instances of `Point`. One could reason about these instances based on Figure 2, along with the modules referred to in that code. To reason about `Point`'s `moveNE` method we would need to consider the specification of the `FigureElement` module since (in JML) methods inherit the specifications of the methods that they override and the method signatures that they implement. But this consideration of `FigureElement` is still modular because `FigureElement` is explicitly referred to by the clause

```
implements FigureElement
```

in the declaration of `Point`.[1] The additional specification for `moveNE` in the `Point` module is combined with the inherited specification from `FigureElement` to form the *effective specification* (i.e., the complete specification that must be satisfied at run-time)

1. In Java every class is implicitly a subclass of `java.lang.Object`. Thus reasoning in Java also requires that one consider `Object`'s specification. However, because it is common to all classes we do not consider this implicit reference to be non-modular.

```
package foal02;
aspect PointMoveChecking {
   private final String MOVE_SW =
      "did you mean to call moveSW()?";
   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
      {
         if (dx < 0 && dy < 0)
            throw new
               IllegalArgException(MOVE_SW);
      }
}
```

Figure 3: An AspectJ module providing advice for `Point`

of `Point`'s `moveNE` method. Rules for combining inherited specifications in JML give the following effective specification for `Point`'s `moveNE` method [24]:

```
  public behavior
    forall int oldx, oldy;
    requires oldx == xCtr && oldy == yCtr
          && dx >= 0 && dy >= 0;
    assignable xCtr, yCtr;
    ensures xCtr == oldx + dx
          && yCtr == oldy + dy
          && \result == this;
    signals (Exception z) false;
 also
  public behavior
    requires dx < 0 || dy < 0;
    ensures false;
    signals (Exception z)
            z instanceof IllegalArgException;
```

JML's `also` keyword combines specification cases; it says that when the precondition of one of the combined cases holds, then the rest of that specification case must be satisfied. So, in addition to the specification inherited from `FigureElement`, this effective specification says that when a client fails to satisfy the original precondition the implementation must throw an `IllegalArgException`. (This inheritance enforces behavioral subtyping [7, 18].)

1.3 Non-modular Reasoning in AspectJ

Next we show that modular reasoning is not a general property of AspectJ by considering an aspect-oriented extension to our previous example. Figure 3 gives an aspect, `PointMoveChecking`, that modifies the behavior of `Point`'s `moveNE` method. `PointMoveChecking` declares a piece of *before-advice*, or code to be executed before traversing a join point into a method body. A *join point* is an arc in the dynamic call graph of a program.[2] The before-advice in `PointMoveChecking` is applicable to each join point where a target object of type `Point` receives a call to the method with signature `FigureElement moveNE(int,int)`. The `target` and `args` keywords are used to give names to the target object and arguments of the method call. (AspectJ also has *after-advice*, executed after traversing a join point out of a method body, and *around-advice*, which applies to the join points into and out of a method body.)

The before-advice in `PointMoveChecking` throws an exception if the absolute value of both arguments in a call to `Point`'s `moveNE` method is less than 0. In AspectJ this advice is applied by the compiler without explicit reference to the aspect from either the `Point` module or a client module; so by definition, modular reasoning about the `Point` module or a client module does not consider the `PointMoveChecking` aspect. Thus, modular reasoning has no way to detect that the effective specification of the `moveNE` method should be changed when the `Point` module and `PointMoveChecking` are compiled together. However, when they are compiled together, then intuitively the behavior of `Point`'s `moveNE` method satisfies the following specification[3].

```
   public behavior
     forall int oldx, oldy;
     requires oldx == xCtr && oldy == yCtr
           && dx >= 0 && dy >= 0;
     assignable xCtr, yCtr;
     ensures xCtr == oldx + dx
           && yCtr == oldy + dy
           && \result == this;
     signals (Exception z) false;
  also
   public behavior
     requires dx < 0 || dy < 0;
     ensures false;
     signals (Exception z)
        z instanceof IllegalArgException;
  also
   public behavior
     requires dx < 0 && dy < 0;
     ensures false;
     signals (Exception z)
        z instanceof IllegalArgException
        && z.getMessage().equals(MOVE_SW);
```

Unfortunately, this behavior is only available to the programmer via non-modular reasoning. That is, in AspectJ the programmer must potentially consider every aspect that refers to the `Point` class in order to reason about the `Point` module. So, in general, a programmer cannot "study the system one module at a time" [23].

1.4 Problem Summary

In a paper from ECOOP 2001, arguing for aspect-oriented programming, Kiczales, *et al.* state [12] (p. 327):

> "We would like the modularity of a system to reflect the way 'we want to think about it' rather than the way the language or other tools force us to think about it."

However, we have seen that the lack of support for modular reasoning can sometimes prevent us from thinking about a system "the way we want to think about it". In AspectJ, tool support is provided to compensate for this lack of modularity. Such tools perform the necessary whole program analysis to direct the programmer to the applicable aspects that affect pieces of a module's source code. Other tools for processing AspectJ source code (e.g., typecheckers, compilers, and optimizers) also require a whole program analysis.

We seek a small set of modifications to AspectJ that obviate the need for this whole program analysis either by the programmer or by supporting tools.

The remainder of this paper is organized as follows. Section 2 gives our proposal for modular reasoning. Section 3 evaluates our proposal. Section 4 discusses some limitations of our proposal and considers separate compilation. Section 5 concludes.

2. A PROPOSAL

We have shown that AspectJ in general does not support modular reasoning; in general the effective specification of a module can only be determined by a whole-program analysis. In this section we propose modifying AspectJ by categorizing aspects into two sorts: assistants and observers. "Observers" are limited in that they may not change the effective specifications of the modules they apply to, "assistants" are not limited in this way. Since observers do not change effective specifications, they preserve modular reasoning even when applied without explicit reference by the modules they

2. Join points in AspectJ are actually more general than what we describe. For example, join points can refer to field references and exception handlers [2]. We leave generalization for future work.

3. We will formalize this intuition in Section 2.

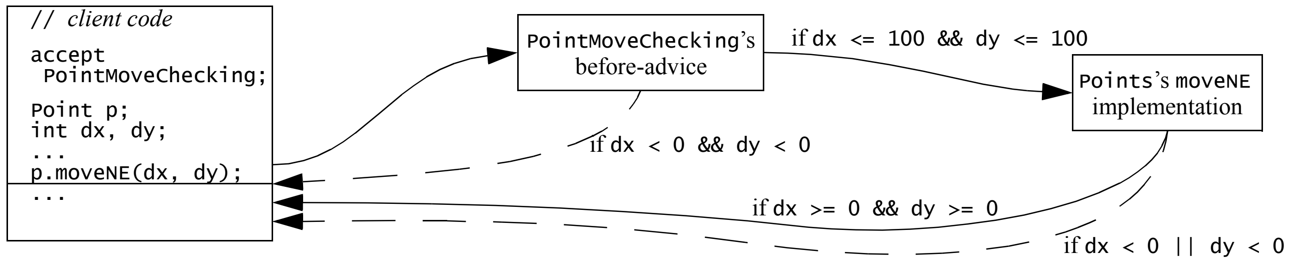

Figure 4: A depiction of the possible control flows of invocations of `moveNE` given that the client module accepts `PointMoveChecking`'s assistance (dashed lines represent exceptional control flow)

observe. Hence observers preserve most of the flexibility of the current version of AspectJ. Because assistants can change the effective specification of the modules to which they apply, to maintain modular reasoning they can only be applied in modules that explicitly reference them. Assistants also require subtle reasoning techniques.

2.1 Assistants

We call aspects that can change the effective specification of a module *assistants*. The `PointMoveChecking` aspect of Figure 3 is an assistant. The term "assistant" is intended to connote a participatory role for these aspects.

What information is needed to modularly reason about behavior when assistants are present? Quite simply, a module must explicitly name those assistants that may change its effective specification or the effective specifications of modules that it uses. We say that a module *accepts assistance* when it names the assistants that are allowed to change its effective specification or the effective specifications of modules that it uses. Assistance may be accepted by:

- the module to which the assistance applies, or
- a client of that module.

AspectJ does not currently include syntax for explicitly accepting assistance. We propose a simple syntax extension for this purpose:

`accept` *TypeName*`;`

where *TypeName* must be a canonical name of an assistant, i.e., a fully qualified name of the package containing the assistant, followed by a "dot" (`.`), followed by the assistant's identifier. Multiple accept-clauses may appear in a single module, following any import-clauses. For example, the `Point` module could accept the `PointMoveChecking` assistance by declaring:

```
accept foal02.PointMoveChecking;
```

Since `Point` (the module implementing the `moveNE` method) accepts `PointMoveChecking`'s assistance, this assistance is applied to every call to `Point`'s `moveNE` method, regardless of the client making the call.

On the other hand, if `PointMoveChecking`'s assistance was accepted by a client module, then that assistance would only be applied to calls from that client to `Point`'s `moveNE` method. Other clients that did not accept the assistance would not have it applied to their calls.

Figure 4 depicts the control flow of an invocation of `moveNE` with `PointMoveChecking`'s assistance. This depiction shows that there are multiple paths by which control may return to the client code, depending on the values of the parameters. The two arrows from the implementation of `moveNE` back to the client code correspond to the two postconditions (normal and exceptional) in the method's specification. Dashed lines indicate exceptional control flow.

2.1.1. Composing Advice Specifications

When a client invokes a method for which either the client or

```
package foal02;
aspect PointMoveChecking {
   private final String MOVE_SW =
      "did you mean to call moveSW()?";
   /*@  public behavior
     @   requires dx > 0 || dy > 100;
     @   ensures true;
     @   signals (Exception z) false;
     @ also
     @  public behavior
     @   requires dx < 0 && dy < 0;
     @   ensures false;
     @   signals (Exception z)
     @     z instanceof IllegalArgException
     @     && z.getMessage().equals(MOVE_SW); @*/
   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
      {
         if (dx < 0 && dy < 0)
            throw new
                  IllegalArgException(MOVE_SW);
      }
}
```

Figure 5: Assistant from Figure 3 with specification added

implementation module has accepted assistance, the behavior of that invocation is based on the sequential composition of the code along a particular control flow path. We can reason abstractly about the possible behavior of the invocation by considering specifications for the method and the assistants. In this subsection we extend JML to specify advice in AspectJ and we show how to modularly reason about the effective specification of a method in the presence of accepted assistance.

A specification language for an aspect-oriented programming language must take possible control flow paths into account. Figure 5 gives another version of the `PointMoveChecking` assistant that adds a specification for the before-advice. In before-advice an ensures-clause gives a normal postcondition, which must hold before control passes to the advised method (or any other applicable advice). We use the ensures-clause in this way since passing control to the advised method is the "normal" behavior for before-advice. So the specification of the before-advice in Figure 5 says that if the advice is entered with `dx > 0` or `dy > 0`, then control flow must pass to the advised method. The implicit frame axiom for this case says that no relevant locations may be assigned when this precondition holds.

The second specification case in Figure 5 says that if the advice is entered with `dx < 0` and `dy < 0` then control flow must return to the caller by throwing an `IllegalArgException` whose message is "`did you mean to call moveSW()?`".

When reasoning about a call to `Point`'s `moveNE` method from the client's perspective we would like to use an effective specification that abstracts away the details of the control flow and intermediate

state transformations. That is, the effective specification from the client's perspective should just concern the preconditions as control flow leaves the client and the postconditions as control flow returns to the client, along with the relevant frame axioms.

Just as the effective behavior along any control flow path is the sequential composition of the code along that path, the effective specification along any control flow path is formed by a kind of sequential composition of the specifications along that path. When a set of paths are in parallel, as in our example, then the effective specification of the set is a kind of parallel composition of the parallel paths' specifications. To formalize these notions we will begin by just considering before-advice and after-returning advice.[4] Then we will use the model to determine the effective specification of `moveNE` in our running example. Later we will sketch extensions to our formal model to accommodate around-advice.

We present our model in two stages. We first describe how to construct a specification composition graph, from the specifications of the implementation module and those of any assistants accepted by that module or the client module. We then describe how the graph is used to determine the effective specification of the invocation.

Constructing a Specification Composition Graph

A *specification composition graph* is a graph whose vertices represent a single method specification, the specifications of all advice applicable to the method (and accepted by the method's implementation module or the client module), and the prestate and poststate from the client's view. The specification composition graph is analogous to the control flow graph for the corresponding code. The specification composition graph is used to determine the possible paths through the advice and method specifications (and hence the code if the implementation is correct). These paths are used to calculate the effective specification.

In general a module may accept assistance from multiple assistants and both a client and an implementation module may accept assistance. The specification composition graph is formed respecting the following order:

1. Apply any before-advice accepted by the client module in the order that it is accepted.
2. Apply any before-advice accepted by the implementation module in the order that it is accepted.
3. Execute the method body.
4. Apply any after-advice accepted by the implementation in the reverse order from which it is accepted.
5. Apply any after-advice accepted by the client module in the reverse order from which it is accepted.

This ordering ensures, for example, that the first assistance accepted by the client is "nearest" to the client and that the last assistance accepted by the implementation is nearest to the implementation on any path.

We will denote this ordering of before-advice, the method, and after-advice by the sequence $\langle a_1, a_2, \ldots, a_n \rangle$ where a_1 through a_{m-1} represent the before-advice, a_m represents the method, and a_{m+1} through a_n represents the after-advice.

4. AspectJ supports three kinds of after-advice. After-returning advice is only applicable when the advised method exits normally. After-throwing advice is only applicable when the advised method exits by throwing an exception. Regular after-advice, without a returning- or throwing-clause, is applicable in either case. To avoid complications in this preliminary proposal we are only considering after-returning advice. It is a simple matter to modify the edge construction algorithm, presented below, to accommodate the other kinds of after-advice.

(For simplicity and modularity we have decided for the present to confine acceptance of assistance to the module in which it is explicitly accepted. For example, if `ColorPoint` is a subclass of `Point`, assistance accepted by `Point` is not automatically applied to invocations of methods declared in `ColorPoint`. On the other hand, if for a particular method `ColorPoint` does not override `Point`'s implementation, then the inherited method carries with it the assistance accepted in the `Point` module. This approach also provides flexibility since the programmer can always add an accept-clause to the subclass module or override a superclass method (gaining assistance in the first case and "shadowing" assistance acceptance in the second). Similar considerations apply for assistance accepted by a superclass module of a client class. Also for simplicity we do not allow interfaces to accept assistance. Future work may reevaluate these decisions.)

Figure 6 gives the specification composition graph for `Point`'s `moveNE` method with assistance from `PointMoveChecking`. It is helpful to refer to this figure while considering the graph construction algorithm.

Formally, a specification composition graph is a directed acyclic graph, $G = \langle V, E \rangle$, where $V = \{start, end\} \cup \{a_i | 1 \le i \le n\}$ is the set of vertices and E is the set of edges.

As Figure 6 shows, each vertex in V, except *start* and *end*, is annotated with the signature of the corresponding method or advice. This information is used reason about the passing of parameters.

To define the edges of the specification composition graph, we first define a function *next* that orders the vertices. Let $next(start) = a_1$, for all $1 \le i \le n-1$, $next(a_i) = a_{i+1}$, and $next(a_n) = end$.

We also need some notation that will be used to label edges in the graph with information from the advice and method specifications. We will use Σ to represent the set of all possible program states, i.e., the set of all legal assignments of values to locations. For each a_i in V, let its specification, $S(a_i)$, be represented by a set of tuples, $S(a_i) = \{S_k(a_i) | S_k(a_i) = \langle Q_k, r_k, f_k, e_k, s_k \rangle, 1 \le k \le p_i\}$, where p_i is the number of cases in the specification and for all k:, $1 \le k \le p_i$, $S_k(a_i)$ represents the kth specification case, in which:

- Q_k represents its set of quantified variables (from `forall`), along with the implicitly bound `\result` variable for methods and after-advice and any variables bound in signals-clauses.
- $r_k : \Sigma \to \text{Bool}$ represents its precondition (`requires`),
- f_k is a set of variables that represents its frame (`assignable`),
- $e_k : \Sigma \to \text{Bool}$ represents its normal postcondition (`ensures`), and
- $s_k : \Sigma \to \text{Bool}$ represents its exceptional postcondition (`signals`).[5]

Each edge in E is represented by a tuple, $\langle \rho, x, y, S_k(x), \sigma \rangle$, with

- $\rho \in \{\nu, \varepsilon\}$ indicating normal (ν) or exceptional (ε) control flow,
- x and y being the beginning and ending vertices of the edge,
- $S_k(x)$ being the kth specification case (as above), and
- $\sigma \in \Sigma$ being the state of the program when control flow traverses that edge.

5. To avoid unnecessary additional complexity we assume each specification in this representation already includes the specifications inherited from its supertypes. Also, typically postconditions are modeled as relations on two states, but we are assuming a form for postconditions that cannot refer to pre-state values.

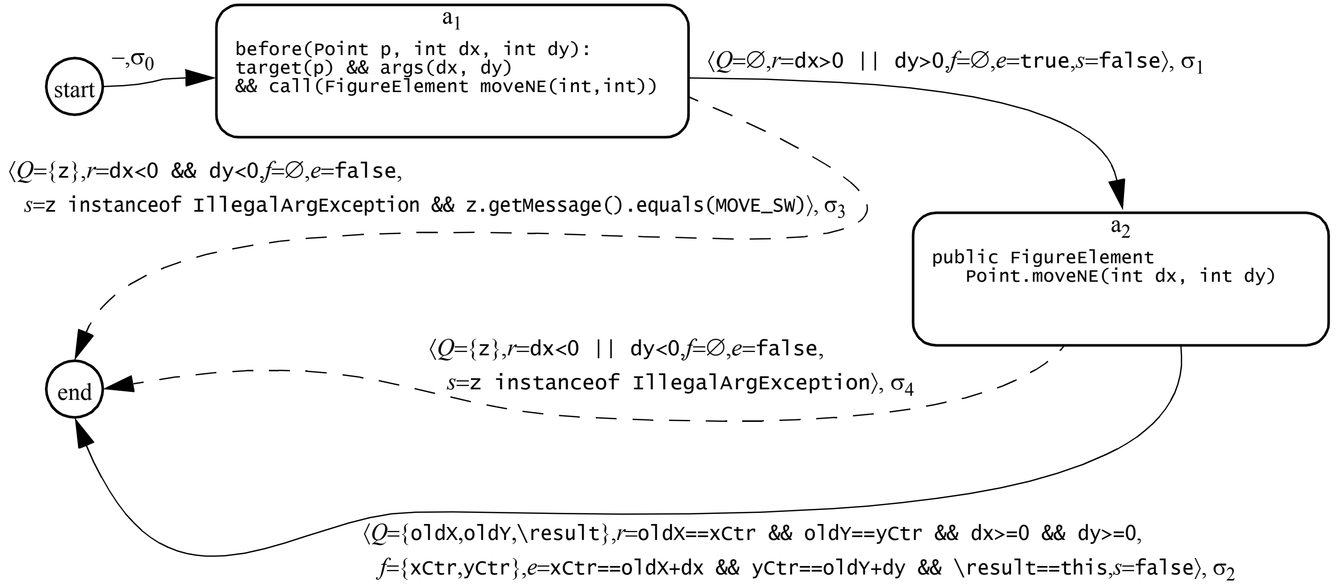

Figure 6: The specification composition graph for `Point`'s `moveNE` method with assistance accepted from `PointMoveChecking` Vertex a_1 corresponds to `PointMoveChecking`'s advice specification, vertex a_2 to `Point`'s `moveNE` specification; edges are labeled with the specification case of the start vertex and the name of the state; exceptional edges are shown with dashed lines.

To model all the possible normal and exceptional control paths, construct E as follows:

1. Let J be an index set of distinct indexes. These will be used to label the state information in the edge tuples.
2. Add a directed edge $\langle \nu, start, next(start), -, \sigma_j \rangle$ to E, where $j \in J$ is an unused index. The empty specification '–' is not used when computing the effective specification.
3. Let $x := next(start)$.
4. Repeat until $x = end$:
   4.1. For each specification case $S_k(x)$ in $S(x)$, if e_k is not `false`, add a *normal edge* $\langle \nu, x, next(x), S_k(x), \sigma_i \rangle$ to E and if s_k is not `false`, add an *exceptional edge* $\langle \varepsilon, x, end, S_k(x), \sigma_j \rangle$ to E, where $i, j \in J$ are unused indices.
   4.2. Let $x := next(x)$.

Figure 6 shows the specification composition graph generated by this algorithm when `Point` accepts assistance from `PointMoveChecking`;

Composing Specifications Along A Path

The specification composition graph, G, contains all the information needed to calculate the effective specification of a method invocation. We first describe how to compose specifications along any single path in G.

Consider a unique path from *start* to *end* in the graph. Because of exceptional return edges this path may not visit every node in the graph. For simplicity of notation we will sequentially renumber the states and for each a_i we will write S_i for the specification case from $S(a_i)$ used on this path. Thus, the path is:

$$\langle \langle \nu, start, a_1, -, \sigma_0 \rangle, \langle \rho, a_1, a_2, S_1, \sigma_1 \rangle, \ldots \\ \ldots, \langle \rho, a_q, end, S_q, \sigma_q \rangle \rangle,$$

where there are q+1 edges on the path. (For a path without exceptional edges $q = n$, otherwise a_q throws the exception.)

To prevent capture of the locally bound variables when composing the specifications, we α-convert the specification cases and related method and advice signatures so that all bound variable names are unique. We reserve the method's formal parameter names for prestate values, so we must α-convert the signature and out-edges of the method vertex. We also reserve the `\result` keyword for the poststate of the effective specification and so all instances of \result in the graph must be α-converted. We will use a fresh variable in signals-clauses of the effective specification. Figure 7 shows the normal control flow path through the specification control graph of Figure 6, after α-conversion.

If a given path is traversed in a program execution, then it must be the case that all the specifications along the path hold. We use this to reason inductively about the path's effective specification.

If control flow enters vertex a_1 then $r_1(\sigma_0)$ holds and the formals in the vertex's signature must be bound to the actual arguments. We use the predicate $\mathrm{bind}(x, y)$ to model the binding of actual arguments, results, or exceptions from vertex x to parameters in the signature of vertex y. This binding is according to the parameter passing semantics of AspectJ and Java, a full definition of which is beyond the scope of this paper. As examples, here are the values of bind for the path in Figure 7, respecting the α-conversion shown there:

- $\mathrm{bind}(start, a_1)$ = `(p1==this && dx1==dx && dy1==dy)`
- $\mathrm{bind}(a_1, a_2)$ = `(this==p1 && dx2==dx1 && dy2==dy1)`
- $\mathrm{bind}(a_2, end)$ = `(this==this && dx==dx2 && dy==dy2 && \result==\result2)`

If control flow leaves vertex a_1 on edge $\langle \rho, a_1, y, S_1, \sigma_1 \rangle$ then the implementation code corresponding to that vertex must ensure that the set of possibly mutated locations is f_1 and that if $\rho = \nu$ then $e_1(\sigma_1)$ holds else if $\rho = \varepsilon$ then $s_1(\sigma_1)$ holds.

If control flow exits vertex a_i on a normal edge $\langle \nu, a_i, a_{i+1}, S_i, \sigma_i \rangle$, $1 \le i \le q$, then the set of possibly mutated locations is $f_1 \cup \ldots \cup f_i$ and the following predicate holds:

$$\mathrm{bind}(start, a_1) \wedge \ldots \wedge \mathrm{bind}(a_{i-1}, a_i) \wedge \\ r_1(\sigma_0) \wedge \ldots \wedge r_i(\sigma_{i-1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_i(\sigma_i)$$

If control flow exits vertex a_i on an exceptional edge, then $i = q$, and the following predicate holds:

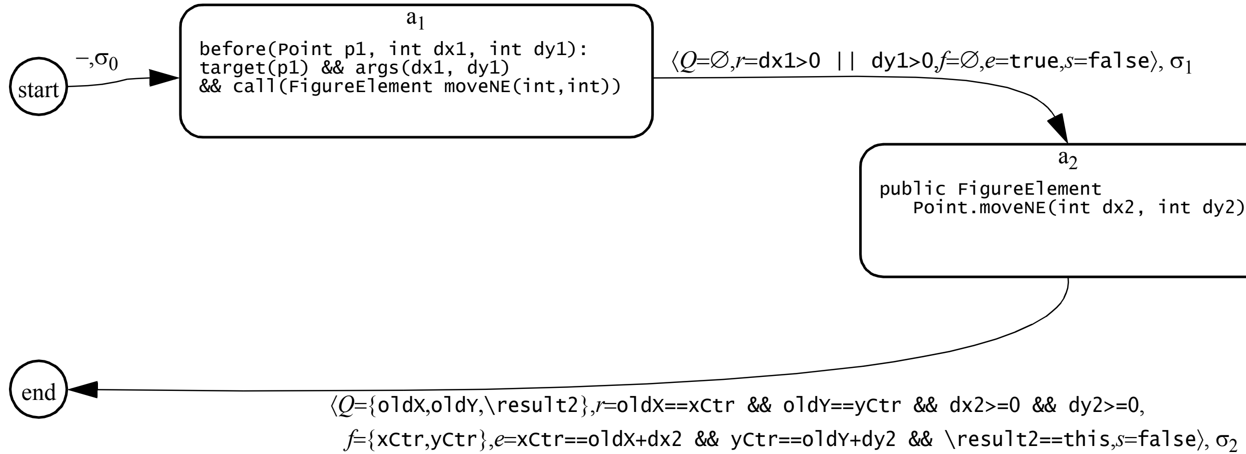

Figure 7: The normal control flow path through the specification composition graph of Figure 6, after α-conversion

$$\text{bind}(start, a_1) \wedge \ldots \wedge \text{bind}(a_{q-1}, a_q) \wedge$$
$$r_1(\sigma_0) \wedge \ldots \wedge r_q(\sigma_{q-1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_{q-1}(\sigma_{q-1}) \wedge s_q(\sigma_q)$$

This predicate involves most of the normal postconditions; these just record what happens along the path before the last edge, which is the only one that throws an exception.

To reason about the effective specification from the client's perspective, we must eliminate the intermediate states from these predicates. One way to do this would be to quantify over the states, like:

$$(\forall \sigma_1, \ldots, \sigma_{i-1} \bullet \text{bind}(start, a_1) \wedge \ldots \wedge \text{bind}(a_{q-1}, a_q) \wedge$$
$$r_1(\sigma_0) \wedge \ldots \wedge r_q(\sigma_{q-1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_{q-1}(\sigma_{q-1}) \wedge s_q(\sigma_q))$$

However, in JML entire states are not directly expressible, so this idea has to be used indirectly by quantifying over intermediate values of each of the variables used in the predicates. Figure 8 gives the general form of the effective specification along any path. The first line of this general form is calculated by this indirect quantification over intermediate values. Let $\hat{y}$ stand for all the free variables[6] (i.e., field names) in specification cases on a path, and let $\hat{T}$ be their corresponding types. We subscript the names in $\hat{y}$ to represent the value of each named variable in the corresponding state. For example, $\texttt{yCtr}_2$ is a variable whose value is that of the field `yCtr` in state σ_2. Similarly, we write $\hat{y}_i$ to represent the vector of all i-subscripted variables, i.e., the vector of values in state σ_i of all variables named in $\vec{y}$.

The second line of Figure 8 gives the explicitly quantified variables of the original specification cases, along with the α-converted parameters from the advice and method signatures.

The requires-, ensures-, and signals-clauses are based on the predicates derived above, with appropriate substitution for the intermediate values of the variables. That is, we write $r_i[\hat{y}:=\overrightarrow{y_{i-1}}]$ for the precondition r_i where for each variable $y \in \hat{y}$, each free occurrence of y is changed to y_{i-1}. We use the same kind of abbreviation for $e_i[\hat{y}:=\vec{y_i}]$.

The general form of the effective specification also must equate the prestate to σ_0 and the poststate to σ_q and include the information provided by the frame axioms. We write $\text{equal}(\vec{y_i},\vec{y_j})$ for the predicate that asserts the equality of the variables in $\vec{y_i}$ and $\vec{y_j}$, using `==` or `.equals` as appropriate for their types, and $\text{notmod}(f,\hat{y},i,j)$ for the predicate that says that variables not listed in the frame f are unchanged; this is defined conceptually as follows (although this quantification is not expressible directly in JML, we can write the equivalent set of conjunctions in any particular case).

$$\text{notmod}(f,\hat{y},i,j) = (\forall y \in \hat{y} \bullet y \notin f \Rightarrow \text{equal}(y_i,y_j))$$

Taken together, we arrive at a single specification case for a single path through the specification composition graph, as Figure 8 shows. Since each possible parallel path is represented by such a specification case, we simply conjoin (using JML's `also` operator) the effective specifications for each path to form the effective specification of the entire invocation (the "parallel composition" alluded to earlier).

Finding the Effective Specification

We can use this formal model to find the effective specification of `Point`'s `moveNE` method with the `PointMoveChecking` assistant.

Consider the path shown in Figure 7. On this path, the free variables are `xCtr` and `yCtr`. Counting the initial state, we need to quantify over 3 states. The effective specification is as shown in Figure 9. Lines from Figure 8 to Figure 9 relate the terms in the general form to the terms in the example. This example specification can be simplified to the following by using transitivity of equality (within clauses), the rule that `false` is the zero of conjunction, and dropping vacuous quantifiers:

```
forall int oldx, oldy;
requires dx >= 0 && dy <= 100
   && oldx == xCtr && oldy == yCtr;
assignable xCtr, yCtr;
ensures xCtr == oldx + dx
   && yCtr == oldy + dy
   && \result == this;
signals (Exception z) false;
```

This is exactly the body of the first specification case arrived at intuitively in Section 1.3. We can analyze the other paths in the graph to calculate the other specification cases. Combining them with `also` yields the full effective specification.

2.1.2. Composition with Around-Advice

Figure 10 gives another assistant, called `PointMoveFixing`. It uses around-advice to change `moveNE` to accommodate negative arguments. Around-advice in AspectJ can execute both before and after the execution of the advised method's body. Unlike before-advice, around-advice can also skip the execution of the advised method's body without throwing an exception. The code (as opposed to the specification) in the body of the advice in Figure 10 illustrates these ideas. If `dx` and `dy` are both non-negative then the statement

6. Though not shown in this paper, JML provides constructs for locally binding names in expressions, such as quantifiers.

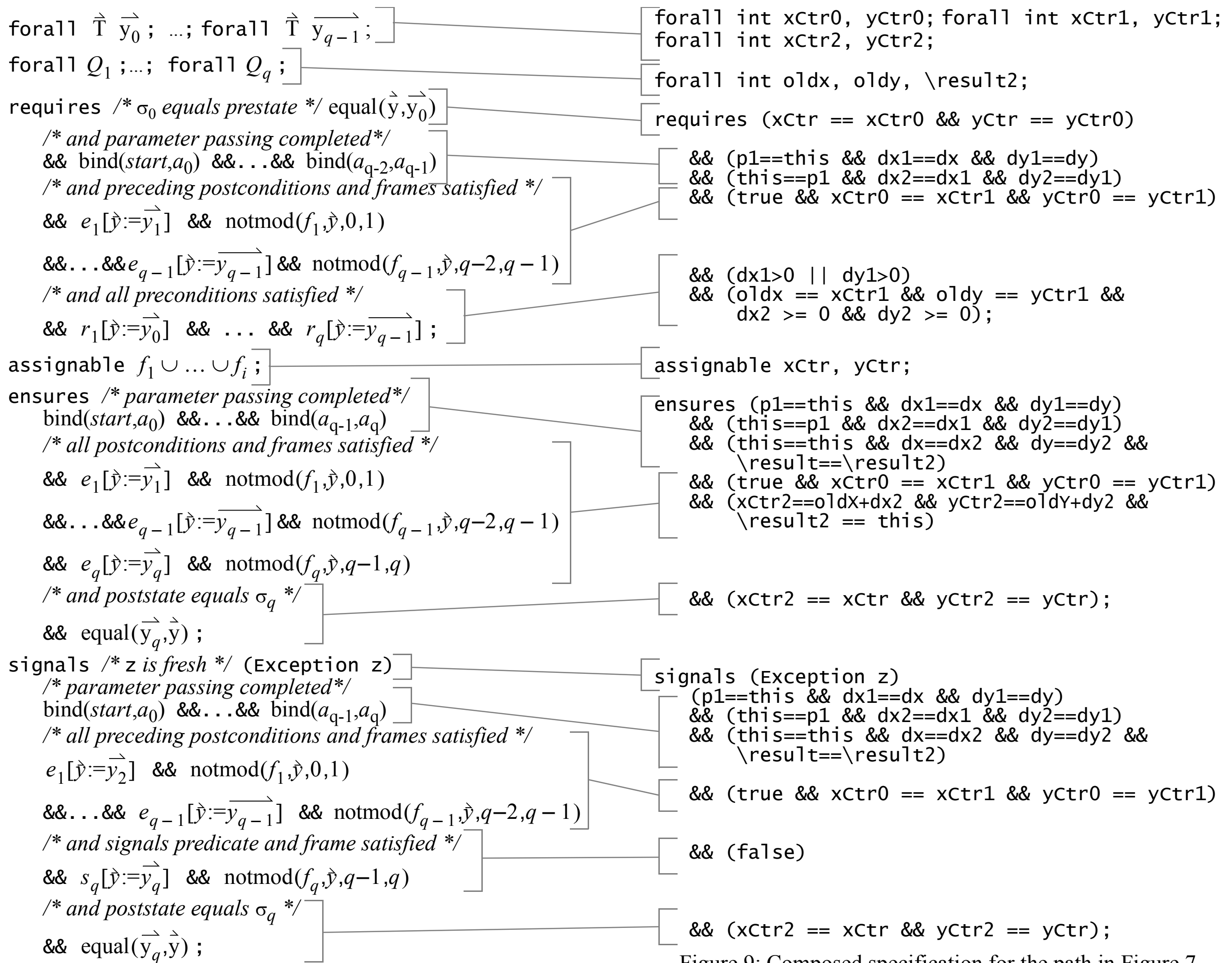

Figure 8: General form of the composed specification for a path

Figure 9: Composed specification for the path in Figure 7
Lines between this and Figure 8 show the correspondence of terms.

```
proceed(p,dx,dy);
```

causes control flow to pass to the original `moveNE` method body with the same arguments as the original invocation. Otherwise, in the else-clause the advice calls the `setX` and `setY` methods on `Point` directly, avoiding the `IllegalArgException` that would be thrown if execution continued into `moveNE`. After the if-statement an acknowledgment message is printed to `System.err`.

Figure 10 also includes a JML specification of the around-advice. As with methods, before-advice, and after-advice, the specification of around advice consists of one or more specification cases joined with the keyword `also`. To specify the additional control flow possible via `proceed` in around-advice, we propose adding the AspectJ proceed-clause to JML as a mechanism for forming compound specification cases. In a specification the proceed-clause joins a specification case called the *before-part*, and a specification case called the *after-part*. The before-part specifies the code executed before proceeding to the original method (and any additional advice if present). The after-part specifies the code executed after returning from the original method (and advice).

The first specification case in Figure 10 (from the beginning up to the `also`) is such a compound specification consisting of before- and after-parts. The case is applicable when `dx` and `dy` are both non-negative, as specified by the requires-clause. In general an ensures-clause in the before-part says that if control flow proceeds to the original method body then the assistant must ensure that the clause's predicate holds. In the example, `ensures true` indicates that control flow can always proceed in this manner. The proceed-clause itself specifies (possibly abstractly) the arguments that will be passed to the original method. A requires-clause in the after-part gives a predicate that can be assumed by the implementation of the after-part. The remainder of the after-part has the usual semantics.

The second specification case, following the `also` keyword in Figure 10, is applicable when at least one of the arguments is negative. The absence of a proceed-clause in this specification case says that control never proceeds to the original method body when this case's precondition is met. The assignable- and ensures-clauses say that control returns to the original client with possible mutation to `p`'s `xCtr` and `yCtr` model fields and the system error stream, and with the given postcondition predicate satisfied.

To reason about effective specifications in the presence of around-advice we would need to extend our formal model. The extension would handle the additional control flow information provided by the proceed-clause. We envision encoding the specification of around-advice with multiple vertices in the specification composition graph. For each piece of around-advice one common vertex would represent the before-parts of all the cases. Separate vertices, one for each case, would represent the after-parts. The edge creation algorithm would require extensions to connect the vertices

```
package foal02;

aspect PointMoveFixing {

   /*@  public behavior
     @    requires dx >= 0 && dy >= 0;
     @    ensures true;
     @   proceed(p,dx,dy);
     @    requires true;
     @    assignable System.err.value;
     @    ensures true;
     @    signals (Exception z) false;
     @ also
     @  public behavior
     @    forall int oldx, oldy;
     @    requires (dx < 0 || dy < 0)
     @            && oldx == xCtr && oldy == yCtr;
     @    assignable p.xCtr, p.yCtr,
     @               System.err.value;
     @    ensures p.xCtr == oldx + dx
     @      && p.yCtr == oldy+ dy;
     @    signals (Exception z) false;
     @*/
   FigureElement around(Point p, int dx, int dy):
      target(p) && args(dx, dy)
      && call(FigureElement moveNE(int,int))
   {
      if (dx >= 0 && dy >= 0) {
         proceed(p,dx,dy);
      } else {
         p.setX( p.getX() + dx );
         p.setY( p.getY() + dy );
      }
      System.err.println("OK");
   }
}
```

Figure 10: An AspectJ module giving around-advice to `Point`

appropriately. The calculation of the composed specification for a given path in the graph would have to account for the expressions in proceed-clauses of the specification.

2.1.3. Summary

We have argued that modular reasoning in aspect-oriented programming languages can be achieved for assistants if we require modules to explicitly accept assistance. We have given a formal model for advice composition that allows us to determine the effective specification of a method. This model also illustrates the reasoning a programmer must undertake even in the absence of formal specifications.

But what impact does our requirement that assistants be explicitly accepted have on the expressiveness of the language? On the one hand, assistants are very expressive in that they are given free rein to change the effective specifications of modules that they assist. On the other hand, requiring assistance to be explicitly accepted dramatically curtails the applicability of assistants. To wit, a common example of the use of aspects is to add tracing capability to an existing program. In a language that just supported explicitly accepted assistance, a programmer would need to make an invasive change to the source code of every module containing a method to be traced (or alternatively, every module calling a method to be traced). We would have gained support for modular reasoning at the expense of modular editing.

2.2 Observers

To resolve this situation we propose that an aspect-oriented programming language should also support a category of aspects that we call observers. An *observer* is an aspect that does not change the effective specification of any other module. Equivalently, an observer may only mutate the state that it owns (in the sense of alias control systems like [20, 21]). It also seems reasonable to allow observers to change accessible global state as well, since a Java module cannot rely on that state not changing during an invocation

```
package foal02;

observer aspect PointMoveTracing {
   private StringBuffer myBuffer =
                             new StringBuffer();

   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
   {
      String message = "Entering Point.moveNE" +
         "(" + dx + "," + dy + ")" + "for " + p;
      myBuffer.append(message);
      System.err.println(message);
   }
}
```

Figure 11: An AspectJ module for tracing method calls.

(modulo synchronization mechanisms). The term "observer" is intended to connote the hands-off role of these aspects. We use the term *observation* to discuss the "advice" in an observer.

For example, Figure 11 gives an observer called `PointMoveTracing`. The `observer` modifier declares that this aspect must not change the effective specification of any other module. This observer mutates its own state by appending to `myBuffer` and mutates the global state by printing to `System.err`. However, it does not change the effective pre- or postconditions of `Point`'s `moveNE` method. `PointMoveTracing` merely observes the arguments to the `moveNE` method and reports them. The arguments are passed on to the method unchanged and the method's results are unchanged.

In addition to cross-cutting concerns like tracing, it seems that observers should be useful for logging and as the observer in the observer design pattern [8] (pp. 293–303).

Because observers do not change the effective specifications of the methods they observe, code outside an existing program can apply an observer to any join point in the original program without loss of modular reasoning. In reasoning about the client and implementation code for a method a maintainer of the original program does not need any information from the observer.

2.2.1. Verifying Observerness

The primary challenge of implementing this part of our proposal lies in determining whether a given aspect is really an observer. We envision a static analysis that conservatively verifies this. This analysis is closely related to the problem of verifying frame axioms. In fact we can think of observers as having an implicit frame axiom that prevents modification of locations that are relevant to the receiver and arguments of the observed method.

The main difficulty with statically verifying this lack of relevant side effects is how to deal with aliasing. For example, suppose we have a logging observer that uses an array to track the elements added to some `Set` object. Suppose `Set` uses an array for its representation. If the observer's array and the `Set`'s array are aliased, we might end up with an element being added to the array twice—possibly violating `Set`'s invariant and changing its effective specification. There is a substantial body of work on alias control that may be useful in attacking this [20, 21].

3. EVALUATION

This section briefly evaluates the practical consequences of our proposal. Because we have not yet had the opportunity to develop applications using our proposed restrictions, our evaluation is limited to a review of existing programs. We first consider the aspect-oriented programming guidelines suggested in the ATLAS case study [10]. Then we survey the example aspects from the AspectJ Programmers Guide [2].

3.1 ATLAS Case Study

In the ATLAS case study [10], the authors proposed several guidelines to make working with aspects easier. These were proposed since they had discovered that (p. 346):

> "[t]he extra flexibility provided by aspects is not always an advantage. If too much functionality is introduced from an aspect it may be difficult for the next developer—or the same developer a few months later—to read through and understand the code base."

One of Kersten and Murphy's suggestions is to limit coupling between aspects and classes to promote reuse. Specifically, they suggest that one should avoid the case where an aspect explicitly references a class and that class explicitly references the aspect, since then the class and aspect are mutually dependent. Such mutual dependencies prevent independent reuse. Is this suggestion problematic for our requirement that modules explicitly accept assistance? No, because the suggestion is concerned with mutual dependence between aspects and classes. Suppose an implementation module, M, accepts assistance from an assistant, A, and A changes M's effective specification. This says nothing about whether M and A are mutually dependent. If A explicitly references M the modules are mutually dependent. However, if A only applies to M because of wildcard-based pattern matching and does not explicitly reference M, then the modules are not mutually dependent. Next, suppose a client module, C, accepts assistance from an assistant, A', and A' only changes the effective specification of modules referenced by C, but does not change C's effective specification. In this case A' and C are not mutually dependent. In sum, programmers can reduce mutual dependency by having clients accept assistance or by limiting explicit references to classes from assistants.

Kersten and Murphy also suggest using aspects as *factories* by having them provide only after-returning advice on constructors. This after-returning advice mutates the state of every object instantiated to change its default behavior. Limiting the aspects in this way restricts the scope of object–aspect interaction. In our proposal a simple assistant can fill the role of a factory aspect.

For aspects that do not act as factories Kersten and Murphy propose three style rules that restrict the use of aspects (pp. 349–350):

> "Rule #1: Exceptions introduced by a weave must be handled in the code comprising the weave. ... Rule #2: Advise weaves must maintain the pre- and post-conditions of a method. ... Rule #3: Before advise weaves must not include a return statement."

These rules are essentially equivalent to our definition of observers in that they prevent aspects from changing the effective specification of the advised method. Though we propose elevating these style rules to the level of statically checked restrictions.

3.2 Dynamic Aspects

The ATLAS case study uses *dynamic aspects*, or the substitution of different aspect code at runtime to modify the behavior of a program. One way to support this technique within the framework of our proposal would be to have modules accept assistance from abstract assistants. Specifications would be associated with these abstract assistants. The various desired behaviors would be implemented as separate assistants, each extending the abstract assistant and implementing its specification. This approach permits modular reasoning. The language would also need a mechanism to support the runtime selection of a particular concrete assistant.

Related to this idea of dynamic aspects is that of a mechanism for combining observers and other modules. In the current version of AspectJ aspects and classes are combined by naming their modules on the command line in a single invocation of ajc, Xerox's AspectJ compiler. Thus combination takes place "outside the language". To support observation of separately compiled programs, we would like to have a mechanism in the language for instantiating observers. It seems that the same language mechanism might support instantiating observers and selecting concrete assistants.

3.3 Impact of Restrictions

We would like to better understand how our restrictions might limit the practical expressiveness of AspectJ. For a preliminary evaluation we use the examples in the AspectJ Programming Guide to see if our restrictions prohibit any recommended idioms. The programming guide's examples can serve this purpose since they "not only show the features [of AspectJ] being used, but also try to illustrate recommended practice" [2] (from Preface). We separate the example aspects into categories based on how we would implement them with our restrictions. An appendix lists the examples by category; we describe the categories here.

Observers. Many of the example aspects clearly meet our definition of observer. To satisfy our restrictions these would only require the new `observer` annotation.

Assistants. Aspects in the examples that could be implemented as assistants can be divided into two kinds. *Client utilities* are used by client modules to change the effective behavior of objects from other modules. The changes in effective behavior do not affect the representation of those objects. To satisfy our restrictions their assistance would have to be explicitly accepted by the clients. In fact, some of the client utility assistants are declared as nested aspects, i.e., aspects declared inside class declarations. These are similar in spirit to explicitly excepted assistance.

There is one example that could be implemented as an assistant but that is not a client utility. This example uses an aspect to separate a simple concern that cross-cuts a single implementation module. The pointcut, or named join point, for this aspect is declared in the implementation class and the aspect explicitly references the implementation class and the pointcut. To satisfy our restrictions the implementation module would have to explicitly accept the assistance, which would create a mutual dependency. However, this example can be considered a bad design since the concern only cross-cuts the one implementation module. This design flaw can be fixed by nesting the assistant in the implementation module, which would also avoid the mutual dependency.

Dynamic Aspects. To satisfy our restrictions some example aspects would require the dynamic aspect mechanisms alluded to in Section 3.2. One such example is a debugging aspect. This aspect would be an observer, except that it provides after-advice to a GUI frame's constructor to add debugging options to the frame's menu bar. To support this pattern with our restrictions requires the mechanisms for dynamic aspects. The GUI frame would have to accept assistance from an abstract assistant, say `AdditionalMenuConcern`, that allowed a concrete assistant, instantiated at runtime, to add to its menu bar. The debugging aspect would become a concrete assistant extending `AdditionalMenuConcern`. The GUI frame could then be instantiated with the debugging assistant or with an assistant that did nothing.

4. DISCUSSION

As presented, our formal model for reasoning about explicitly accepted assistance does not accommodate advice that applies to join points other than those for method invocations. It seems a simple matter to extend the model to accommodate some other kinds of join points, such as those for field access or exception handling. However, it is not clear whether our model can accommodate dynamic context join points [2], like `cflow(`*pointcut*`)`, which rely on runtime information for applicability tests. It seems that advice

on dynamic context join points can only be modularly reasoned about if this advice is confined to observations. There is one aspect in the AspectJ examples we studied, the `Registry.RegistrationProtection` aspect of the spacewar example, that uses a dynamic context join point with advice that changes the effective specification of the advised method. This example is not supported by the current work.

Because of the generality of aspects without our restrictions and limitations of the target Java Virtual Machine (or *JVM*) [17], AspectJ currently requires whole-program compilation [12]. In our proposal, because assistance is explicitly accepted, it is a simple matter to support separate compilation for modules that accept assistance; the compiler just weaves it into the accepting modules.

On the other hand, observers present interesting challenges for separate compilation. On the surface, since observers do not change the effective specifications of other modules, it should be possible to separately compile them. And indeed this is true—except for the issue of dispatching to observers. The generality of observers means that they can potentially be dispatched to from any join point.

Thus, the only obstacle to separate compilation of AspectJ programs given our restrictions is that of dispatch to observers. Others have suggested that separate compilation for AspectJ is possible using techniques such as specialized class loaders or modified virtual machines [12] (p. 343). With our proposed restrictions the scope of the problem is reduced, likely making it easier to implement these solutions.

5. CONCLUSIONS

To summarize, we have shown that with a few simple modifications AspectJ can support modular reasoning. Our proposal separates aspects into two categories, assistants and observers, which provide complementary features. Assistants are extremely powerful, but require subtle reasoning techniques and are limited in their applicability to maintain modular reasoning. Observers are less powerful but are easy to reason about and are broadly applicable. This broad applicability is achieved by placing heavier burdens on the type system.

A preliminary evaluation showed that for many cases our modifications to the language provide sufficient flexibility. However, we also noted that there is a need for some mechanism to support dynamic aspects.

The other major open problem for our proposal is statically checking that aspects declared as observers meet our definition, as discussed in Section 2.2.1. To attack this problem we propose:

- developing an aspect-oriented calculus for investigating these ideas in a formal setting, and
- developing and proving sound a type-system for the calculus that statically enforces our proposed restrictions on observers.

Other future work on the problem of modular reasoning for aspect-oriented programming languages includes:

- refining our proposed specification constructs for AspectJ and formalizing their semantics, perhaps using something like the refinement calculus [3], and
- investigating behavioral subtyping and formal techniques for verification of aspect-oriented programs.

We are also interested in demonstrating the utility and effectiveness of our ideas by:

- programming non-trivial systems using our restrictions,
- integrating the proposed restrictions into AspectJ, and
- understanding the potential benefits of our restrictions for separate compilation, static analysis, and optimization.

In this paper we have focused on adding support for modular reasoning to the AspectJ language. Future work will also investigate the relevance of our proposal to other aspect-orientation programming languages and techniques, such as composition filters [4], adaptive methods [16], and multidimensional separation of concerns as embodied by Hyper-J [22, 25].

ACKNOWLEDGMENTS

We would like to thank Yoonsik Cheon, Todd Millstein, Markus Lumpe, and Robyn Lutz, for their helpful comments on a draft of this paper. The work of Leavens was supported in part by the US National Science Foundation grants CCR-0097907 and CCR-0113181. The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea.

APPENDIX

Table 1 below lists the aspects from the examples directory of the Version 1.0.1 release of AspectJ[7]. The second column of the table gives the categorization of each example based on the categories of Section 3.3.

Table 1: Example Aspects and their Categories

Example	Category
telecom/TimerLog	observer
tjp/GetInfo	observer
tracing/lib/AbstractTrace	observer
tracing/lib/TraceMyClasses	observer
tracing/version1/TraceMyClasses	observer
tracing/version2/Trace	observer
tracing/version2/TraceMyClasses	observer
tracing/version3/Trace	observer
tracing/version3/TraceMyClasses	observer
bean/BoundPoint	client utility
introduction/CloneablePoint	client utility
introduction/ComparablePoint	client utility
introduction/HashablePoint	client utility
observer/SubjectObserverProtocol	client utility
observer/SubjectObserverProtocolImpl	client utility
spacewar/Display.DisplayAspect	client utility
spacewar/Display1.SpaceObjectPainting	client utility
spacewar/Display2.SpaceObjectPainting	client utility
telecom/Billing	client utility
telecom/Timing	client utility
spacewar/EnsureShipIsAlive	assistant[a]
coordination/Coordinator	dynamic
spacewar/Debug	dynamic
spacewar/GameSynchronization	dynamic

7. Available from `http://www.aspectj.org`.

Table 1: Example Aspects and their Categories

Example	Category
spacewar/RegistrySynchronization	dynamic
spacewar/Registry.RegistrationProtection	unsupported[b]

a.The EnsureShipIsAlive aspect considered to be a poor design in the discussion of Section 3.3.
b.The aspect, Registry.RegistrationProtection, uses dynamic context join points, which aren't supported by the current work.

REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[2] AspectJ Team, the. The AspectJ programming guide. Available from http://aspectj.org/doc/dist/progguide/index.html, Feb. 2002.

[3] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[4] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[5] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. Available from www.multijava.org.

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[7] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[10] M. A. Kersten and G. C. Murphy. Atlas: A case-study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 340–352, Denver, CO, November 1999. ACM.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, Oct. 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin Heidelberg, June 2001.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, Aug. 2001. See verb|www.cs.iastate.edu/ leavens/JML.html|.

[15] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[16] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, Oct. 2001.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Co., Reading, MA, second edition, 2000.

[18] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[19] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[20] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's PhD Thesis. Available from http://www.informatik.fernuni-hagen.de/import/pi5/publications.html.

[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[22] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, Oct. 2001.

[23] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[24] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, Aug. 2001.

[25] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.