Data-flow and Control-flow Analysis of AspectJ Software for Program Slicing

Guoqing Xu
Ohio State University

Atanas Rountev
Ohio State University

Abstract

We propose an approach for program slicing of AspectJ software, based on a novel data-flow and control-flow program representation. The representation is built at the source-code level and, unlike previous work, captures the semantic intricacies of various pointcut designators, multiple advices per joint point, dynamic advices, exceptions, and general flow of data to, from, and between advices. We also present algorithms for dependence analysis, system dependence graph construction, and slicing of AspectJ programs. These algorithms are built on top of the proposed representation, and take into account the complex flow of data and control due to aspect-oriented features. We present an experimental study on 37 programs, using our AJANA *analysis framework which is based on the abc AspectJ compiler. The results show that the representation can be built efficiently, that it is superior to an approach based on the woven bytecode, and that slicing is both faster and more precise. These findings strongly indicate that the proposed general approach is a promising solution for slicing and similar analyses of AspectJ software, in various tools for program comprehension, change impact analysis, program integration, software testing, and software debugging.*

1 Introduction

Program slicing is a popular static analysis technique with a wide range of applications for program comprehension, change impact analysis, program integration, reverse engineering, software testing, and software debugging [5, 9, 6, 7, 16]. The increasing popularity of aspect-oriented programming presents a serious challenge to analysis designers: how should existing program slicing techniques be generalized to handle aspect-oriented features?

A key component of program slicing approaches is the program's interprocedural control-flow graph (ICFG). In previous work [20] we proposed an approach for building the ICFG of an AspectJ program. This effort focused on control-flow semantics, and did not answer a critical question: how should the *data-manipulating effects* of ICFG nodes be modeled and analyzed? In this paper we propose a solution to this problem. As a result, it becomes possible to perform dependence analysis, program slicing, and a range of similar interprocedural static analyses for AspectJ.

Since the executable code of an AspectJ program (produced by an AspectJ compiler) is pure Java bytecode, an obvious approach is to apply directly existing analyses for Java to the bytecode. Of course, this requires an analysis to build and preserve a map that associates the effects of each entity in the bytecode back to those of its corresponding entity in the source code. However, as pointed out by our previous work [20], there is significant discrepancy between the Java/AspectJ source code and the woven Java bytecode. This makes it extremely hard to establish such a map. For example, calls to *proceed* in around-advices can be interpreted as calls to different methods at different join point shadows, and can sometimes even be interpreted as the inlining of the body of the crosscut method when that body is sufficiently simple. Furthermore, the correspondence between source-level and bytecode-level entities is specific to the weaving compiler being used; different compilers (or different version of the same compiler) can create completely different mappings.

An alternative approach is to perform analysis on the source code or some suitable intermediate form derived from the source code; this is the approach taken in our work. Analyses are complicated by the semantic complexity of the pointcut types, by situations where multiple advices may apply at the same join point, and by the existence of dynamic advices which match a join point statically, but may or may not match it at run time. We propose a novel program representation which makes explicit the data that is exposed during the interactions between advices and base code, and associates this data with the appropriate ICFG nodes and edges. Unlike previous work, our approach handles challenging situations such as multiple advices that apply at the same join point, and the existence of dynamic advices. It precisely represents the interactions occurring in join points that can be described by 15 types of pointcut designators, out of a total of 17 defined in the AspectJ language (except for *cflow* and *cflowbelow*).

We show how the proposed representation can be used to construct the *system dependence graph* (SDG) of an As-

pectJ program. SDGs are commonly used for program slicing and other techniques that require dependence information. We present an interprocedural dependence analysis and an interprocedural slicing algorithm. This contribution provides the first general definition of a slicing approach for AspectJ which takes into account the full complexity of interactions due to aspect-oriented features.

The representation and the slicing algorithm have been implemented in our AJANA (**A**spect**J** **ana**lysis) framework, built as an extension of the *abc* AspectJ compiler [2]. We performed an experimental evaluation of the proposed techniques. Our study indicates that, compared to an approach based on the woven Java bytecode, (1) the program representation and the SDG have significantly smaller sizes, especially for programs that contain around-advices; (2) more precise slices can be computed using our SDG; and (3) significant reduction in analysis running time can be achieved.

This work makes the following specific contributions:

- *Program representation.* We propose a data-flow and control-flow program representation for AspectJ programs, as basis for interprocedural dependence analysis, program slicing, and similar static analyses. The representation is built at the source-code level and captures the semantic intricacies of 15 pointcut designators, in the presence of multiple advices per joint point, dynamic advices, exceptions, and general flow of data to, from, and between advices.
- *Dependence analysis and slicing.* We present an approach for interprocedural dependence analysis, SDG construction, and program slicing for AspectJ. The approach is built on top of the program representation, and provides a general slicing technique which takes into account the full complexity of data flow and control flow due to aspect-oriented features.
- *Experimental evaluation.* We present a study for 37 programs, using our AJANA analysis framework. The results show that the cost of building the representation is practical, that ICFG and SDG sizes are reduced compared to analysis of woven bytecode, and that slicing is both faster and more precise. These findings strongly indicate that the proposed general approach is a promising solution for slicing and similar analyses of AspectJ software, in various tools for program comprehension, change impact analysis, program integration, software testing, and software debugging.

2 Example and Background

For illustration we will use a modified version of the *bean* example from the AspectJ distribution. Figure 1 shows classes *Point* and *Demo*. Aspects are used to implement an event firing mechanism, by invoking *propertyChange* when a change occurs. A field *support* and a method *addPropertyChangeListener* are introduced in *Point* by aspect *BoundPoint*, at lines 3–7 in Figure 2. For brevity, helper class *PropertyChangeSupport* is not shown.

```
1  class Point {
2   int x = 0, y = 0;
3   int getX() { return x; }
4   int getY() { return y; }
5   void setRectangular(int nX, int nY) {
6     setX(nX); setY(nY);
7   }
8   void setX(int nX) { x = nX; }
9   void setY(int nY) { y = nY; }
10  String toString() { println("X="+x+",Y="+y); }
11 }
12 class Demo implements PropertyChangeListener {
13  void propertyChange(PropertyChangeEvent e) {...}
14  static void main(String[] args) {
15    Point p = new Point();
16    p.addPropertyChangeListener(new Demo());
17    p.setRectangular(5,2); println("p = " + p);
18    p.setX(6); p.setY(3); println("p = " + p);
19  }
20 }
```

Figure 1: Running example, classes *Point* and *Demo*.

A *join point* in AspectJ is a well-defined point in the execution that can be monitored. We classify the join point types in AspectJ into four categories: (1) initialization, including both object initialization and class initialization, (2) method/constructor call and execution (3) field getting and setting, and (4) exception handling. For a particular join point, the textual part of the program executed during the time span of the join point is the *shadow* of the join point [4]. We classify shadows in two categories: *statement shadows*, for which the program entity that is advised is a statement (e.g., a call), and *body shadows*, where the advised entity is the body of a method or constructor.

A *pointcut* selects ("picks out") one or more join points by imposing run-time restrictions on the basic join point types, and optionally exposes some of the values from the execution context. AspectJ defines 17 types of primitive pointcut designators. We classify them into three categories: *join point selector*, *run-time condition specifier*, and *data exposer*. Table 1 shows the classifications of join point types, pointcut designator types, and their relationship. A pointcut is dynamic if it has a property of *run-time condition specifier*; otherwise, a pointcut is a static one. A combined pointcut is dynamic if at least one of its component pointcuts is dynamic; otherwise it is static.

► *Example.* Aspect *BoundPoint* from Figure 2 defines two pointcuts: *setter* and *getterX*. The shadows of the join points picked out by *setter* are the five call sites at lines 6, 17, and 18 in Figure 1. The shadow of the join point picked out by *getterX* is the body of method *getX*. Both pointcuts are dynamic because they include *target(p)*, which picks out only run-time objects that are instances of *Point*. ◄

```
1  aspect BoundPoint {
2   // add a field 'support' to class Point
3   PropertyChangeSupport Point.support =
4     new PropertyChangeSupport(this);
5   void Point.addPropertyChangeListener
6   (PropertyChangeListener l)
7     { support.addPropertyChangeListener(l); }
8   void firePropertyChange(Point p, String property,
9   double oldv, double newv) {
10    p.support.firePropertyChange(property,
11    new Double(oldv),new Double(newv));
12  }
13  // ====== pointcuts ======
14  pointcut setter(Point p):
15    call(void Point.set*(*)) && target(p);
16  pointcut getterX(Point p):
17    execution(void Point.getX(*)) && target(p);
18  // ===== advices for pointcut 'setter' =====
19  before(int x, Point p)
20  : setter(p) && args(x) { // before1
21    if (x < 0) println("Bad set*");
22  }
23  after(Point p) : setter(p) { // after1
24    println("Return from set*");
25  }
26  void around(Point p) : setter(p) { // around1
27    int oldX = p.getX(); proceed(p);
28    firePropertyChange(p,"1",oldX,p.getX());
29  }
30  void around(Point p) : setter(p) { // around2
31    Point p1 = new Point(); proceed(p1);
32    firePropertyChange(p,"2",p.getX(),p1.getX());
33  }
34  // ===== advices for pointcut 'getterX' =====
35  before(Point p) : getterX(p) { // before2
36    println("Start getX");
37  }
38  after(Point p) returning (int x)
39  : getterX(p) { // afterReturning1
40    println("Return from getX: " + x);
41  }
42 }
```

Figure 2: Running example, aspect *BoundPoint*.

An *advice declaration* consists of an advice kind (before, after, etc.), a pointcut, and a body of code forming an *advice*. Whenever multiple advices apply at the same join point, precedence rules determine the order in which they execute [3]. We refer to an advice associated with a dynamic pointcut as a *dynamic advice*.

► *Example.* In Figure 2, *before1*, *after1*, *around1*, and *around2* may apply at the same join point. Similarly, *before2* and *afterReturning1* may apply at the same join point. The execution order of the first four advices is *before1,around1[around2[cs,after1]]* where *cs* is the advised call site, and brackets enclose advices that are invoked by the call to *proceed* in the preceding around-advice. ◄

2.1 Control-Flow Representation

In previous work [20] we considered the problem of regression test selection for AspectJ software, and proposed a control-flow representation for identifying the differences between two versions of the same program. Using essentially the same approach, one could construct the interprocedural control-flow graph (ICFG) of an AspectJ program. The rest of this subsection describes some of the details of the ICFG control-flow representation. Figure 3 shows a subset of the ICFG for the example from Figures 1 and 2.

JP category	PC designator	PC category
initialization	*initialization*	JP selector
	preinitialization	JP selector
	staticinitialization	JP selector
call/execution	*call*	JP selector
	execution	JP selector
	adviceexecution	JP selector
field get/set	*get*	JP selector
	set	JP selector
excpt handling	*handler*	JP selector
	within	condition specifier
	withincode	condition specifier
	this	condition specifier
	target	data exposer, condition specifier
	args	data exposer, condition specifier
	if	condition specifier
	cflow	condition specifier
	cflowbelow	condition specifier

Table 1: Classification of join points and pointcuts.

This is the first work to define a control-flow representation for AspectJ which handles multiple advices at the same join point, the existence of dynamic advices, and exceptions. An ICFG contains (1) standard CFGs that model the control flow within Java classes, within aspects and across boundaries between aspects and classes through non-advice method calls, and (2) *interaction graphs* (IGs) that model the interactions between methods and advices at join points. An IG is built for each statement shadow.

► *Example.* Consider shadow *p.setX(6)* at line 18 in Figure 1. In the absence of aspect-oriented features, this call would be represented by two ICFG nodes: a *call-site* node and a *return-site* node. Interprocedural edges would connect the call-site node with the start node of *setX*, and the exit node of *setX* with the return-site node. For this example, advices *before1*, *around1*, *around2*, and *after1* apply at the corresponding join point. In the ICFG the call is represented by an artificial method *ph_root* (*ph_* stands for "placeholder") which represents the top-level logic associated with the run-time processing of the join point. ◄

Handling of multiple advices. The precedence rules can be used to build a helper data structure, the *advice nesting tree*, to represents the run-time advice nesting. Each tree level contains at most one around-advice, which is the root of all advices in the lower levels. With each around-advice A the tree associates (1) a possibly-empty set of before-advices and after-advices, (2) zero or one around-advices, and potentially (3) the actual call site that could be invoked

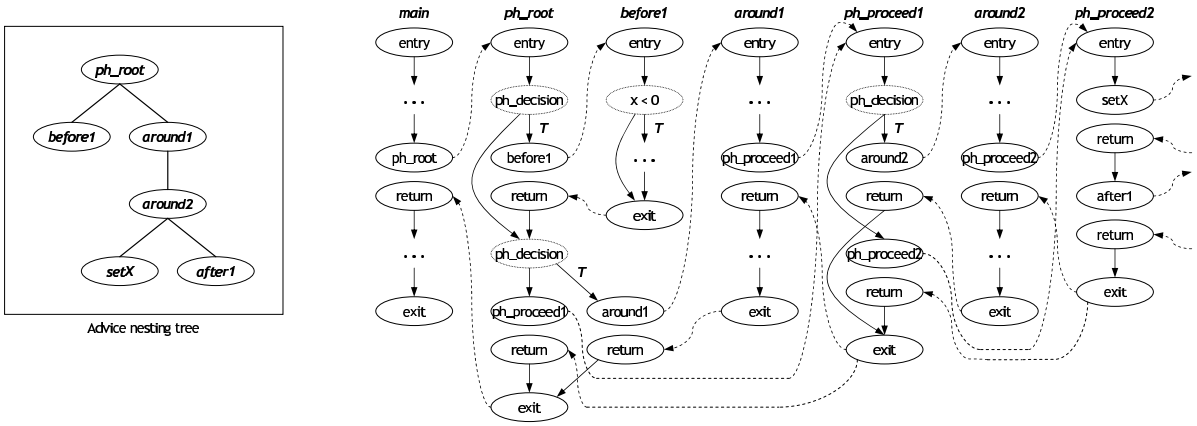

Figure 3: Advice nesting tree and partial interaction graph for shadow *p.setX(6)*.

by the call to *proceed* in A. These advices and the call site appear as if they were nested within A. The nesting tree for *BoundPoint* at shadow *p.setX(6)* is shown in Figure 3.

Nodes at one level of the tree are invoked by the call to *proceed* in the around-advice in the upper level of the tree. A placeholder method *ph_proceed* is used to represent the proceed call in an around-advice. This method contains calls to all children advices, including the crosscut call site. For example, *ph_proceed1*, which represents the proceed call in *around1*, contains a call to *around2*, whose proceed call is in turn represented by *ph_proceed2*, which then calls *setX* (the shadow) and *after1*. The top-level *ph_root* method corresponds to the root of the advice nesting tree.

Handling of dynamic advices. For dynamic advices that may or may not be invoked at the join point, the ICFG uses an artificial *ph_decision* node to guard the execution. For a dynamic advice, the "true" edge leaving this node goes directly to its call-site node — that is, if the run-time condition evaluates to true, the advice will be invoked. For a non-around-advice, the "false" edge goes to the call-site node for the next advice that could be invoked in the current method. For an around-advice, the "false" edge goes to a call-site node for its corresponding *ph_proceed* method, meaning that if this advice is not invoked, the advices that are nested within it will still be invoked.

Handling of exception-triggered after advices. An exception thrown by an advice or the crosscut method could trigger the invocation of an after or after-throwing advice. Due to space constraints, this paper will not discuss the treatment of exceptions. Our approach and implementation do handle exceptional constructs; details of this handling will be available in a companion technical report.

3 Data-Flow Representation

The representation outlined above captures the control-flow semantics of aspect-oriented interactions. In order to apply existing analyses for dependence analysis and program slicing, the key problem becomes: *what data can flow in the interaction graph at a join point, and how can it be represented?* This section presents our novel technique for building an ICFG-based data-flow representation by associating data with related ICFG nodes and edges. This is the first work to consider the full generality of data flow to, from, and between advices in AspectJ. Our goals are to (1) make variables that are defined/used during the interaction explicit for the placeholder methods that reference them; (2) associate with *ph_decision* nodes the variables that are used to make run-time decision about dynamic advices; (3) expose a minimum set of variables, without introducing any unnecessary variables and extra helper variables in IGs; and (4) keep a single CFG for each advice declaration, without replicating graphs for different advice applications.

In this paper we focus the discussion on *call* join points (see Table 1), which are the most common join points in AspectJ practice. Due to space limitations, we do not discuss the other types of join points, although our implementation also handles them. For call joint points, the key goal is to make explicit the flow of data by creating formal parameters and actual parameters associated with ICFG nodes, in order to enable analysis based on this data-flow representation.

Declarations for placeholder methods and non-around-advices. For a call join point, the variables that can be referenced are the actuals $\langle a_1, a_2, \ldots, a_n \rangle$ at the call site, as well as the receiver object reference a_0 if the crosscut call site is an instance invocation. Given an IG, each placeholder method is parameterized with formals $\langle f_0, f_1, f_2, \ldots, f_n \rangle$ that match the actuals of the shadow. Each such method is static and has the same return type as the return type of the method called at the shadow. For example, *ph_root* in Figure 3 has signature *void ph_root (Point arg0, int arg1)* where *arg0* corresponds to the receiver object reference at *p.setX(6)* and *arg1* corresponds to the actual parameter. For a non-around-advice that is called by a placeholder method, a *static void* declaration is used with the same formals as in the source code declaration. For example, *before1* in Figure 2 is declared as *void before1(int arg0, Point arg1)*. An

after-returning-advice has one last formal parameter corresponding to the returned value specified in the advice declaration. For example, the signature for *afterReturning1* from Figure 2 is *void afterReturning1(Point arg0, int retval)*.

Call sites for non-around-advices. Placeholder methods contain calls that invoke non-around-advices; to enable static analysis, the appropriate actuals must be associated with these calls. For an advice that has declared parameters, there have to be one or more data exposer pointcut designators associated with it (i.e., *target* or *args* from Table 1). A helper function $cf(f_i, p)$ (short for "corresponding formal") can be defined, where p is the type of pointcut and f_i is a formal parameter in the advice declaration. Given f_i, cf is used to obtain the corresponding formal of the caller placeholder method. This formal should be used as the actual for f_i at the call site. Thus, for an advice adv declared as

adv*(t,p_1,...,p_n) : target(t) && args(p_1,...,p_n) && ...*

a call site of the following form is created

$$adv(cf(t, target), cf(p_1, args), .., cf(p_n, args))$$

For example, in Figure 3, the call to *before1* inside *ph_root* has the form *before1(arg1,arg0)* where *arg0* and *arg1* are formal parameters of *ph_root*.

The IG also contains a call for the shadow. The formals of the caller placeholder method are used as actuals (or receiver) of this call. If the shadow is an assignment, a *return$val* local variable in the placeholder method is assigned the return value at the call. For example, for the invocation of *setX* inside *ph_proceed2* in Figure 3, the call is *arg0.setX(arg1)* where *arg0* and *arg1* are the formals of *ph_proceed2*. For a call to an after-returning-advice, an additional actual parameter is needed. In the caller placeholder method, *return$val* should have already been assigned the return value of a call to the crosscut method (or to another *ph_proceed* method); this local is used as the last actual parameter at the call to the after-returning-advice.

3.1 Handling of Around-Advices

The complication in handling an around-advice is that its formals are dependent on the crosscut method and on other advices that are invoked within the around-advice. First, the around-advice must have access to all actual parameters of the shadow in order to call the crosscut method or a *ph_proceed* method. For example, consider *ph_proceed2* in Figure 3, which is called from within *around2*. Because the formals for *ph_proceed2* are *(Point arg0, int arg1)*, *around2* has to take at least these two types of formals, although it has only one declared formal (line 26 in Figure 2). Second, the formals needed by an around-advice could be different for different shadows that the advice matches, because such shadows may call different methods due to the use of wild cards (*) in the pointcut definition. Consider again *around2* defined in Figure 2. Due to the *setter* pointcut, *around2* it can crosscut both *setX* and *setRectangular* calls. For shadows that call *setX*, the formal parameter types needed by *around2* are *(Point, int)*, but for shadows that call *setRectangular* the types needed are *(Point, int, int)*.

One possible approach is to replicate the CFG of an around-advice for each shadow that the advice matches, and to create the method declaration and the call site for the advice on per-shadow basis. Hence, one can have shadow-specific placeholder methods and around-advices, and globally unique non-around-advices. In fact, this approach is being used by the *abc* compiler [2]. However, this violates one of our design goals: to keep one CFG per advice, without replicating the CFG per advice application. Such replication could result in an explosion in the number of ICFG nodes, and therefore may introduce significant overhead for subsequent static analyses. In fact, our experiments showed that for some benchmarks containing around-advices that match every call site, the approach used by *abc* can cause the size of the program to double after compilation.

Declarations and call sites for around-advices. We propose a different approach which does not replicate around-advices. For each such advice we consider all shadows that the advice matches, and construct a globally-valid list $global\_params$ that includes parameters that are required at each shadow. A companion map sp (short for "shadow position") maps each shadow to the starting position of its corresponding parameters in $global\_params$.

► *Example.* Consider *around2* and two of the shadows it applies to: *p.setX(6)* and *p.setRectangular(5,2)* at lines 18 and 17 in Figure 1. The corresponding actuals have types *(Point, int)* and *(Point, int, int)*. Hence, $global\_params$ is *(Point arg0, int arg1, Point arg2, int arg3, int arg4)*. For *p.setX(6)* the starting position in $global\_params$ is 0, and for *p.setRectangular(5,2)* the position is 2. There are three more shadows at which this advice applies; for brevity, we omit the details related to these shadows. ◄

List $global\_params$ together with the originally declared parameters of the around-advice define the formals for this advice. A last formal is added for a *decision value* indicating the shadow where the advice is currently applied. For example, the signature for *around2* becomes *void around2(Point arg0, int arg1, Point arg2, int arg3, int arg4, Point p, int dv)* where *arg0* through *arg4* come from $global\_params$, *p* is the original declared parameter (line 30 in Figure 2), and *dv* is the decision value.

In each placeholder method where an around-advice is called, the call for that advice has non-trivial actuals only for (1) formal parameters corresponding to the currently-active shadow, as defined by the positions in map sp, (2) the advice's original formal parameters, and (3) the last formal parameter *dv*. For formals corresponding to the shadow, the formals of the caller placeholder method are used as actuals. For the advice's orig-

inal formal parameters, the actuals are constructed similarly to calls to non-around-advices. A unique shadow ID is used as the actual for formal parameter *dv*. For example, for shadow *p.setX(6)*, the call for *around2* is *around2(arg0,arg1,null,0,0,arg0,987)* where *arg0* and *arg1* are the formals of *ph_proceed1*, which contains the call site. The first two actuals correspond to the shadow's parameters, while the next-to-last actual corresponds to the declared formal *p* of *around2*. Similarly, for *p.setRectangular(5,2)*, the call site is *around2(null,0,arg0,arg1,arg2,arg0,567)*. The last actuals in these two call sites (*987* and *567*) are unique IDs for the corresponding shadows.

Around-advice calls to placeholder methods. Because there is a single CFG for an around-advice, the advice should be able to call different *ph_proceed* methods for different shadows. A pair of call-site and return-site nodes is created for each placeholder method that the advice could call; the original call to *proceed* is replaced with this group of calls. A placeholder decision node is created to represent the selection of a placeholder method to be called, and formal *dv* is associated with this decision node. Essentially, this representation is equivalent to a switch statement.

The actual parameters for the calls to the placeholder methods can be defined similarly to the actuals for calls to around advices — the formals corresponding to the shadow are identified in the around-advice's list of formals, and are used as actual parameters at the call site. However, additional transformations are necessary: actuals that correspond to the originally declared parameters of the around-advice must be replaced with the actuals for the original call to *proceed* in this around-advice. This is necessary for cases when *proceed* is called with values other than the formals of the around-advice, in which case the new values need to be propagated to the *ph_proceed* methods as well.

► *Example.* Advice *around2* needs to call *ph_proceed2* at shadow *p.setX(6)* and another placeholder method (e.g., named *ph_proceed2_2*) at shadow *p.setRectangular(5,2)*. The following pseudocode illustrates the representation:

```
static void around2(Point arg0, int arg1,
Point arg2, int arg3, int arg4, Point p, int dv) {
   Point p1 = new Point();
   switch(dv) {
      // used to be ph_proceed2(arg0,arg1)
      case 0: ph_proceed2(p1,arg1);
      // used to be ph_proceed2_2(arg2,arg3,arg4)
      case 1: ph_proceed2_2(p1,arg3,arg4);
   }
}
```

The "used to be" comments show the calls before taking into account the fact that the original call to *proceed* (line 31 in Figure 2) uses *p1* and not *p* as an actual parameter. ◄

After creating the call sites, redundant formals that are not used by the advice can be eliminated. For example, after such a removal, the signature of *around2* becomes *void around2(int arg1, int arg3, int arg4, Point p, int dv)*.

3.2 Data for Placeholder Decision Nodes

For a placeholder decision node that guards a call-site node for an advice, we need to associate the data that contributes to the decision making. There are two kinds of such decision nodes: (1) a shadow-based selection decision node in an around-advice (e.g., *switch(dv)* in the example from above), and (2) a decision node that guards a dynamic advice. For the first kind, the associated data is the formal *dv*. For the second kind, there has to be a run-time condition specifier associated with the guarded dynamic advice. As shown in Table 1, there are eight kinds of such designators in AspectJ. For *within* and *withincode*, one can statically determine if the pointcut matches. For *target* and *args*, which are also data-exposer designators, the needed data are the parameters specified in the pointcut. A *this* designator indicates that the receiver object at the shadow must be an instance of the type specified by the pointcut; the needed data is a reference to the object. The *if* pointcut can only reference parameters introduced by data-exposer pointcuts. For *cflow* and *cflowbelow*, there does not exist an explicit value that affects decision making. Our tool currently ignores these two kinds of pointcuts; future work will have to develop static analysis techniques for handling them.

► *Example.* Consider the placeholder decision node in *ph_root* that guards the call to *around1* (shown in Figure 3). The run-time condition specifier in *around1*'s *setter* pointcut (line 15 in Figure 2) is *target*. Therefore, the data that should be associated with this decision node is the formal of *ph_root* that corresponds to the receiver object — that is, formal parameter *arg0*. ◄

4 Slicing AspectJ Programs

The novel data-flow and control-flow representation described earlier can be used to define the first general slicing algorithm for AspectJ software. It is important to note that this algorithm takes full advantage of the information in the program representation: for example, it correctly and precisely tracks the data dependencies at a joint point at which multiple advices are applied (taking into account the order of advice execution), as well as the control dependencies due to the AspectJ semantics for run-time decisions and their guarded dynamic advices. Due to space constraints, we describe the algorithm at a high level and omit a number of technical details.

The dependence analysis used in our technique is representative of flow- and context-sensitive dataflow analysis algorithms; all such algorithms require the information encoded in our representation. In the current implementation we do not model the effects of library calls; a precomputed library summary is needed for interprocedural analyses [14], and the problem of library summary generation is outside of the scope of this paper. The implementation uses a simple alias analysis, and determines call and return edges

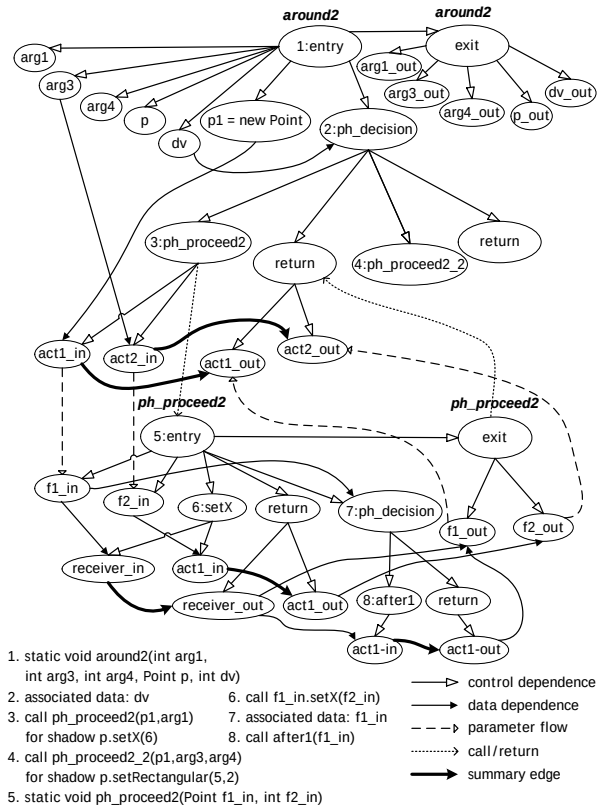

Figure 4: Partial SDG for *around2*.

using class hierarchy analysis.

Building the system dependence graph (SDG). Given the program representation, the SDG of an AspectJ program can be constructed relatively easily. The SDG contains data-dependence and control-dependence edges between ICFG nodes, together with special nodes and edges to represent the effects of calls. We use the algorithm by Horwitz et al. [10] to build the SDG. A key feature of this algorithm is the computation of summary edges that represent transitive dependencies along same-level interprocedurally-valid ICFG paths [12]. We construct such edges using a bottom-up interprocedural dependence analysis which takes full advantage of the rich semantic information embedded in the program representation.

A decision node is created for each virtual call site; each control-dependence edge leaving this node goes to a call-site node for a possible method that could be invoked at run time. For these and other placeholder decision nodes, the variable associated with the node is considered used (i.e., read). The technique from [1] is used to handle exceptions.

► *Example.* Figure 4 illustrates a partial SDG for the running example, again with focus on shadow *p.setX(6)*. Since relevant data is associated with placeholder decision nodes, the algorithm can take into account the data dependencies between such nodes and the nodes that define the data. For example, the proceed-selection decision node 2 in the SDG is data dependent on the formal *dv*. Similarly, decision node 7 is data dependent on formal *f1_in*, which is a reference to the receiver object of the crosscut call site, because this node represents the *target* pointcut that guards the execution of dynamic advice *after1*. ◄

Program	#LOC	#Versions	#Methods	#Shadows
bean	296	7	40	11
tracing	1059	7	44	32
telecom	870	7	96	19
quicksort	111	3	18	15
nullcheck	2991	5	196	146
lod	3075	3	220	1103
dcm	3423	4	249	359
spacewar	3053	1	288	369

Table 2: Analyzed programs.

Slicing AspectJ software. Graph-reachbility-based slicing [10] can be directly used on the SDG. Each statement in the source code, except calls to *proceed*, corresponds to one ICFG node. For computing a forward or backward slice for any non-*proceed* statement, the slicing algorithm is executed starting from the corresponding ICFG node. A call to *proceed* in an around advice may correspond to a group of call-site nodes in the ICFG, each one of which calls a *ph_proceed* method for one shadow. In this case a slice is computed for each call-site node, and the union of the resulting slices forms the complete slice of the call to *proceed*.

5 Empirical Evaluation

To evaluate the proposed techniques, we performed a study which focused on the following questions: What are sizes of the ICFG and the SDG built with our approach, compared to the ones built from the woven bytecode? What is the effect on program slicing with respect to slice size and computation time, compared to slicing on the woven bytecode? What is the cost of building the representation?

Implementation. The data-flow and control-flow representation and the program slicer were implemented in our AJANA analysis framework. The framework is built on top of the *abc* AspectJ compiler [2]; details on the weaving performed by *abc* can be found in [4]. AJANA uses the Jimple intermediate representation produced by the static weaving component of the compiler, before the actual advice weaving process starts. At this point the inter-type fields and methods introduced by aspects have been added to their host classes, and static shadows have been identified, which significantly facilitates our analysis.

Analyzed programs. Our study used the eight AspectJ programs shown in Table 2. The first seven program were used in our previous work to evaluate a technique for regression test selection [20]; in that work, the original version of each program was used as basis to create several modified versions. The last benchmark was taken from the AspectJ

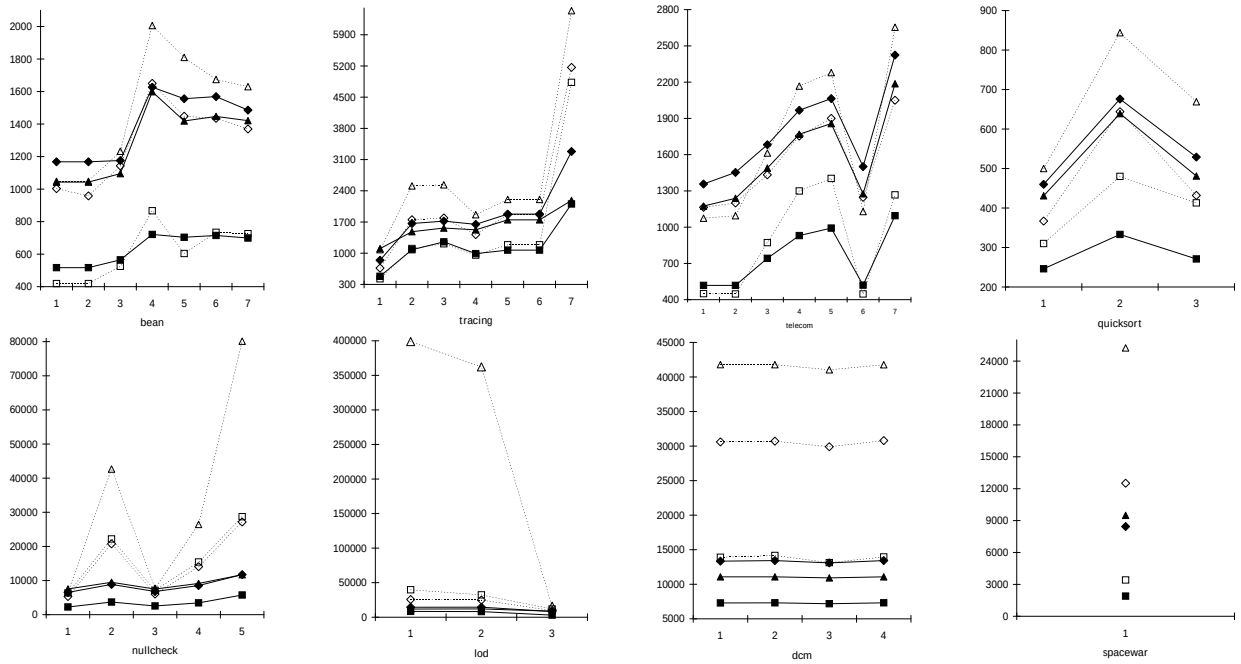

Figure 5: Number of ICFG edges □/■, control dependence edges ◇/⧫, and data dependence edges △ /▲ constructed from woven bytecode (□, . . .) and by AJANA (■, . . .).

example package. This group of benchmarks have also been used by other researchers to evaluate their work on test case generation [18] and on measuring the performance of AspectJ programs [8]. For each program, Table 2 shows the number of lines of code, methods, and shadows in the original version, plus the number of modified versions. Considering the different versions of the same program, the study used a total of 37 experimental subjects.

Study 1: ICFG/SDG size and cost. Our first study investigated the sizes of the ICFG and the SDG. Figure 5 shows a comparison between the graphs built with the proposed representation and the graphs constructed from the woven bytecode. The figure shows the number of ICFG edges, the number of SDG control-dependence edges, and the number of SDG data-dependence edges. We consider edges rather than nodes, because the number of edges is a more important factor that affects the running time of subsequent analyses such as program slicing and change impact analysis.

For some simple program versions (e.g., for benchmarks *bean* and *telecom*), our approach produces more ICFG edges. We investigated these occurrences, and determined that in these versions some advice bodies contain only a few statements, and the weaving process inlines these bodies at their shadows. In these cases, inlining eliminates interprocedural edges that are explicit in our representation. Except for such cases, the number of ICFG edges in our representation is lower — for example, by at least a factor of two for *tracing.7*, *nullcheck*, *lod*, *dcm*, and *spacewar*.

For most versions, the SDGs built by our technique have slightly more control-dependence edges, due to the artificial decision nodes in the representation. On the other hand, the number of SDG data-dependence edges constructed from the woven bytecode is often dramatically higher that the corresponding number in AJANA. For example, this number is more than 7 times higher for *nullcheck.5*, and more than 30 times higher for two versions of *lod*. We inspected the versions with significant differences, and found that all of them contain around-advices, and many of these advices crosscut every call site in the base program. In such cases, any analysis based on data dependencies is likely to incur significant overhead unless it employs our representation.

The representation can be built in practical time. For example, for *nullcheck*, *lod*, *dcm* and *spacewar*, which are the largest among the eight benchmarks, the analysis ran in 156.4, 159.4, 417.75, and 250 ms respectively, whereas the corresponding times for ICFG building from the woven bytecode were 122.0, 122.0, 535.5, and 110 ms. Even though the building of our representation can be slower than building the ICFG for the woven bytecode, these differences will be offset by savings in subsequent analyses; such savings are evident in our study of slicing, presented below.

In addition to size and cost, another significant benefit of our representation is its independence from the particular weaving techniques used to create the final Java bytecode. An analysis built with this representation depends only on the semantics of the AspectJ language, but not on the specific low-level implementation details of the weaving compiler. Such independence has critical advantages when relating the analysis results back to the original program (e.g., in tools for comprehension and testing), and when consider-

Program	Ver	Size (nodes)	RelRat (%)	Time (sec)
bean	v4	367 / 618	98.2 / 54.7	2.2 / 2.6
tracing	v7	829 / 1443	98.7 / 77.2	5.8 / 24.2
telecom	v7	387 / 230	98.5 / 85.3	2.0 / 1.3
quicksort	v2	309 / 325	98.0 / 44.5	0.3 / 0.8
nullcheck	v2	836 / 5852	97.8 / 37.7	15.9 / 542.7
	v5	3313 / 7203	96.1 / 46.0	21.1 / 762.1
lod	v1	926 / 3593	97.2 / 64.9	32.0 / 105.1
	v2	652 / 3623	97.9 / 60.1	21.2 / 956.7
dcm	v2	1444 / 8654	98.2 / 44.3	19.7 / 1011.7
spacewar	v1	169 / 1687	97.6 / 68.6	3.9 / 62.8

Table 3: Slice size, relevance ratio, and slicing time.

ing different weaving compilers or different versions of the same compiler. Furthermore, an analysis may employ techniques that are specialized for precise semantic modeling of aspect-oriented features; such techniques cannot be used at the lower abstraction level of the woven bytecode.

Study 2: Program slicing. Our second study investigated the proposed slicing algorithm. One of the major motivations of computing a slice for a program entity is to understand its dependencies on other program entities. Consider a slice S and the set of all SDG nodes included in S. From the point of view of a programmer, only nodes that correspond to entities from the original program source code (both from the Java base code and from the advices) are of any relevance for program comprehension through slicing. Therefore, an interesting question is the following: how many SDG nodes in S correspond to statements in the original AspectJ code? We will refer to the number of such nodes, relative to the total number of nodes in S, as the *relevance ratio* of the slice.

We implemented and ran the standard slicing algorithm [10] on the SDG built by AJANA and on the SDG built from the woven bytecode. For each benchmark, we choose the most complex versions, in the sense that they contained the most complex advice interactions. We computed a slice for each node in the SDGs of these versions, and determined the relevance ratios for all slices. Table 3 summarizes the results of this experiment. Column "Ver" shows the program versions that were used. Column "Size" contains the average number of nodes in a slice; a slash "/" separates the result obtained with our representation from the one obtained with the woven bytecode. Using a similar format, column "RelRat" shows the average value of the relevance ratios for the computed slices, and column "Time" shows the total time of the computation, including SDG building.

For calculating the relevance ratios of the slices, the key is to determine the set of SDG nodes that do *not* have corresponding statements in the AspectJ source code. For slices based on our representation, these nodes are all placeholder decision nodes as well as all call-site and return-site nodes for placeholder methods. For slices based on the woven bytecode, it is not obvious how to identify such nodes, due to the difficulty in establishing a map between the source code and the woven code. As an approximation, we define this set to contain all nodes in compiler-introduced methods. In fact, the relevance ratios shown in Table 3 may be too high for the bytecode-based approach, because even in methods whose declarations are not changed by the compiler, there may be compiler-introduced statements.

Clearly, our technique achieves significantly better relevance ratios, which means that the slices it computes are much closer to the original AspectJ source code. Furthermore, for all programs except *telecom*, smaller slices are built and the running time for SDG building and slicing is reduced. Especially for large programs (such as the last four), our technique achieves impressive time savings. We manually inspected the woven bytecode for *telecom* and determined that inlining done by the weaving process was the reason for the reduced cost of SDG construction and slicing.

Conclusions. For analyzing AspectJ software, especially larger AspectJ applications, the source-code-based representation proposed in this paper is practical to build, easier to understand, contains significantly fewer nodes and edges, and can dramatically speed up subsequent program slicing and similar interprocedural analyses.

6 Related Work

Analysis and testing for AOP software. The *abc* compiler group [2] developed the AspectBench Compiler [4] for AspectJ which provides a variety of static analyses and optimizations. Their work focuses on optimizations of the generated bytecode to reduce execution overhead, whereas the focus of our work is representation and analysis at the source-code level, abstracting away compiler-specific details. We implemented AJANA as an extension to the *abc* compiler, building the ICFG between the static weaving phase and the advice weaving phase.

Rinard et al. [13] classify the interactions between methods and advices, in order to enable developers to recognize interactions that support modular reasoning and to focus on the causes of potentially non-modular interactions, by employing a pointer and escape analysis [17]. Pioneering work by Zhao [23, 21, 22, 24] defines program representations for a variety of testing and analysis tasks for aspect-oriented programs. These initial results do not consider more complex situations such as multiple advices per join point or dynamic advices, and do not perform experimental evaluation. Our work builds on these earlier advances and defines several new theoretical and experimental contributions.

A body of work considers testing of aspect-oriented programs. Souter et al. [15] develop a test selection technique based on concerns. Xu and Xu [19] present a specification-based testing approach for aspect-oriented programs. Xie and Zhao [18] describe a wrapper class synthesis technique

and a framework for generating test inputs for AspectJ programs. These previous efforts focus on techniques to enable the testing of aspect-related features, whereas our work develops general representation and analysis of data flow and control flow, which could facilitate future work on data-flow and control-flow-based testing of AspectJ programs.

Our previous work proposed a static control-flow model for AspectJ software [20] which serves as the basis for the ICFG used in this paper. This earlier work developed a new graph-traversal algorithm for selecting regression tests for AspectJ software. However, this approach did not include any data-flow representation or analysis.

Dataflow analysis and program slicing. Interprocedural dataflow analyses have been investigated extensively. The work presented in this paper opens up the opportunity to apply these techniques to AspectJ software. In particular, a natural direction for future work is to consider graph-reachability-based analyses for IFDS [12] dataflow problems. In fact, the dependence analysis used in our work is an instance of this general category.

There is a large body of existing work on program slicing [5, 9, 6, 7, 16]. Slicing algorithms for aspect-oriented programs have been proposed in [22, 11]. These important early contributions do not consider the full complexity of the flow of data and control at join points, or the generality of AspectJ language features. Furthermore, it is not clear how interprocedural dependence analysis would be performed, and such analysis is not implemented or evaluated. Our work continues this line of investigation by defining and evaluating experimentally a general data-flow and control-flow representation which can be used for dependence analysis and a variety of other interprocedural analyses (e.g., the general family of dataflow analyses from [12]).

7 Conclusions

This paper describes a general approach for constructing a static data-flow and control-flow representation for AspectJ software, for the purposes of dependence analysis, program slicing, and similar interprocedural analyses. Our experiments clearly show that, compared to analysis of the woven bytecode, this representation is much better suited as foundation for subsequent static analyses. The proposed approach creates promising opportunities for a large body of future work on adapting existing analyses to AspectJ and on designing novel AspectJ-specific analyses, for use in various tools for program comprehension, impact analysis, and software testing and debugging.

References

[1] M. Allen and S. Horwitz. Slicing Java programs that throw and catch exceptions. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 2003.

[2] *AspectBench Compiler*. `abc.comlab.ox.ac.uk`.

[3] *AspectJ Compiler*. `www.aspectj.org`.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. `abc`: An extensible AspectJ compiler. In *Int. Conf. Aspect-Oriented Software Development*, 2005.

[5] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[6] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[7] A. De Lucia. Program slicing: Methods and applications. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, 2001.

[8] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, 2004.

[9] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Tran. Programming Languages and Systems*, 12(1):26–60, 1990.

[11] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Int. Conf. Software Maintenance*, 2004.

[12] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. Principles of Programming Languages*, pages 49–61, 1995.

[13] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Symp. Foundations of Software Engineering*, pages 147–158, 2004.

[14] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *Int. Conf. Compiler Construction*, pages 2–16, 2006.

[15] A. Souter, D. Shepherd, and L. Pollock. Testing with respect to concerns. In *Int. Conf. Software Maintenance*, 2003.

[16] F. Tip. A survey of program slicing techniques. *J. Programming Languages*, 3:121–189, 1995.

[17] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[18] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Int. Conf. Aspect-Oriented Software Development*, pages 190–201, 2006.

[19] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Int. Conf. Aspect-Oriented Software Development*, pages 180–189, 2006.

[20] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *Int. Conf. Software Engineering*, 2007.

[21] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Int. Workshop on Principles of Software Evolution*, pages 108–112, 2002.

[22] J. Zhao. Slicing aspect-oriented software. In *Int. Workshop on Program Comprehension*, pages 251–260, 2002.

[23] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Int. Computer Software and Applications Conf.*, page 188, 2003.

[24] J. Zhao and M. Rinard. System dependence graph construction for aspect oriented programs. In *MIT-LCS-TR-891*, 2003.