AJANA: A General Framework for Source-Code-Level Interprocedural Dataflow Analysis of AspectJ Software *

Guoqing Xu Atanas Rountev
Ohio State University
{xug,rountev}@cse.ohio-state.edu

Abstract

Aspect-oriented software presents new challenges for the designers of static analyses. Our work aims to establish systematic foundations for dataflow analysis of AspectJ software. We propose a control- and data-flow program representation for AspectJ programs, as basis for subsequent interprocedural dataflow analyses. The representation is built at the source code level and captures the semantic intricacies of various pointcut designators, multiple applicable advices per joint point, dynamic advices, and general flow of data to, from, and between advices. We also propose two dataflow analyses for AspectJ software: (1) a novel object effect analysis based on a flow- and context-sensitive must-alias analysis, and (2) a dependence analysis used for constructing the system dependence graph for slicing, refactoring, change impact analysis, etc. Both analyses are representative of a general category of dataflow analyses referred to as interprocedural distributed environment (IDE) problems. The two analyses are built on top of the proposed representation, and take into account the complex flow of control and data due to aspect-oriented features. We present a study of the proposed techniques on 37 program versions, 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 it enables analyses that are both faster and more precise. These findings strongly indicate that the proposed approach is a promising candidate for a foundation upon which various interprocedural analyses for AspectJ can be designed and built.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Languages

Keywords Dataflow analysis, interprocedural analysis, AspectJ

1. Introduction

Interprocedural dataflow analysis is a form of static analysis that plays an important role in various software tools. Hundreds of different analyses have been used for software understanding and maintenance (e.g., for program comprehension, slicing, change impact analysis, automatic transformations, etc.). The semantic information produced by dataflow analysis is also used as an essential component of tools for software verification and testing. In addition, dataflow analysis plays an important role for performance improvements through compiler optimizations. A large body of existing work has considered the theoretical foundations for dataflow analysis (e.g., [27, 22, 26]) as well as specific analyses for imperative languages, and more recently, for object-oriented languages.

The increasing popularity of aspect-oriented programming presents two serious challenges to the designers of dataflow analyses. First, how should existing analyses be generalized to handle aspect-oriented features? Second, what kinds of new analyses can contribute to better understanding, maintenance, testing, verification, and optimization of aspect-oriented software? A critical step in attacking these challenges is the definition of systematic general foundations for dataflow analysis of aspect-oriented languages. The goal of our work is to make novel advances towards achieving this goal for AspectJ software.

A key component of dataflow analyses is the program's interprocedural control-flow graph (ICFG) and the dataflow functions associated with edges in this graph [27, 22]. In previous work [34] 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 represented and modeled in dataflow analyses? In this paper we propose a solution to this problem. This solution, for the first time, makes it possible to define explicitly the lattice and functions for dataflow problems for AspectJ software. The proposed approach can serve as the starting point for a large body of work on adapting existing analyses to AspectJ and on defining new analyses for AspectJ-specific problems.

Since the executable code of an AspectJ program (produced by an AspectJ compiler) is pure Java bytecode, an obvious approach is to directly apply existing analysis techniques for Java to the bytecode. Of course, this approach requires an analysis to build and preserve a map that associates the data-flow 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 [34], 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 even different version of the same compiler) can create completely different mappings.

* This material is based upon work supported by the National Science Foundation under grant CCF-0546040.

AOSD '08 March 31–April 4, 2008, Brussels, Belgium.

An alternative approach is to perform dataflow analysis on the source code or some suitable intermediate form derived from the source code; this is the approach taken in our work. Such source-code-level (SCL) analysis has several advantages over bytecode-level (BCL) analysis. First, SCL analysis produces more relevant results that can be more easily understood and interpreted by human programmers. For example, given a particular variable in an advice, clients of a points-to analysis are interested only in the objects *in the source code* that this variable may point to. BCL analysis would return a set of objects including those introduced by the compiler, which creates significant comprehension obstacles for programmers. Second, SCL analysis can be performed before weaving and therefore can produce information about the effects of advices on the base code. The information may include, for example, the purity of an advice (absence of side effects), the write/read effects of advices on objects that flows from/to the base program, the sequence of methods invoked by advices on a particular receiver object, etc. This information allows a software verification tool to statically check certain properties, such as typestate-based object protocols or specifications. BCL analysis, on the other hand, must be performed after code is woven, and therefore is incapable of providing such information. Finally, SCL analysis is, in most cases, faster than BCL analysis on large aspect-oriented programs. For example, in our experiments, SCL slicing is faster than BCL slicing for 9 out of the 10 experimental data points.

SCL dataflow analysis is complicated by the semantic complexity of the various 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. At present, there does not exist a general and complete treatment of these issues. We propose a program representation which makes explicit the data that is exposed during the interactions between advices and the base code, and associates this data with the appropriate ICFG nodes and edges. The representation considers cases where multiple advices apply at the same join point, as well as the presence 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 illustrate the uses of the representation by designing two dataflow analyses: (1) a novel *object effect analysis* based on a flow- and context-sensitive must-alias analysis, and (2) a *dependence analysis* used for constructing the system dependence graph for slicing, refactoring, change impact analysis, etc. Both analyses are representative of interprocedural distributed environment (IDE) dataflow analysis problems [26]. The IDE class is a general category of problems, examples of which are copy-constant propagation and linear-constant propagation [26], object naming analysis [25], 0-CFA type analysis [14, 31, 15], and all IFDS (interprocedural, finite, distributive, subset) problems [26] such as reaching definitions, available expressions, live variables, truly-live variables, possibly-uninitialized variables, flow-sensitive side-effects [7], some forms of may-alias and must-alias analysis [23], and interprocedural slicing [16]. Through our approach, this rich set of existing interprocedural dataflow analyses becomes possible to apply to AspectJ software.

The effect analysis is used to build regular expressions for each shadow (i.e., base code corresponding to a join point) to summarize the effects that the combination of advices applicable at the shadow can have on the objects that flow to the advices from the base code. Given a set of objects passed into advices at a shadow, the effect information includes a sequence of read and write operations on fields of these objects, and a sequence of methods invoked on them. The regular expressions produced by this analysis can be directly used to analyze the interactions between the base code and the advices, as well as inter-advice interactions. For example, the analysis can be employed to verify the interactions of advices with the the base code against properties considered harmful in aspect-oriented software development, such as the writing, in an advice, of an object that is read in the base code. As another example, object protocols and typestate properties can be directly inferred and/or verified from the analysis output. The proposed analysis techniques are novel contributions for static analysis of AspectJ software for program understanding and verification.

The system dependence graph [16] has traditionally been used for program slicing and other techniques that require interprocedural dependence information. As a second example of an interprocedural dataflow analysis, we define a form of dependence analysis and the corresponding slicing algorithm. Slicing of aspect-oriented programs should not be done on the woven code, because otherwise the resulting slices could contain a lot of code that is not human understandable, which in turn would complicate, rather than simplify, program understanding tasks. Earlier approaches for slicing of aspect-oriented software [36, 38, 17] have various limitations — these analyses were built upon simplified program representations that did not capture the full complexity of the problem. Our work defines a more general solution and provides new insights based on an extensive experimental evaluation.

The proposed representation and the two dataflow analyses have been implemented in our AJANA (**A**spect**J** **ana**lysis) framework, built as an extension of the `abc` AspectJ compiler [1]. We performed an experimental evaluation of the proposed techniques. Our study indicates that, compared to the BCL analysis (1) the effects of multiple advices (applying at a shadow) on the incoming objects can be precisely computed; (2) the program representation and the SDG have significantly smaller sizes, especially for programs that contain around-advices; (3) more precise slices can be computed using our SDG; and (4) significant reduction in analysis running time can be achieved with our approach.

The key contribution of this work are:

- **Program representation.** We propose a technique for building a program representation for AspectJ programs, as basis for subsequent interprocedural dataflow analyses. The representation is built at the source-code level and captures the semantic intricacies of pointcut designators in the presence of multiple applicable advices per joint point, dynamic advices, and general flow of data to, from, and between advices.
- **Effect analysis and summary generation.** At the core of the object effect analysis is a must-alias analysis used to identify variables that definitely refer to the objects of interest. Based on the must-alias information, we generate regular expressions for each object passed into an advice from the base code, to summarize the effects that the advice can have on the object.
- **Dependence analysis and slicing.** We present an approach for dependence analysis, SDG construction, and slicing of AspectJ programs. The approach is built on top of the proposed representation, and takes into account the complex flow of data due to aspect-oriented features.
- **Experimental evaluation.** We present an experimental study on 37 program versions drawn from 8 base programs, using our AJANA framework. The results show that an effects summary can be efficiently computed, that the cost of building the representation and running the analysis 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 our approach is a promising candidate for a foundation upon which a rich variety of interprocedural dataflow analyses for AspectJ can be designed and built.

```
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  void reset() {x = 0; y = 0;}
11  String toString() { println("X=" + x + ", Y=" + y); }
12 }
13 class Demo implements PropertyChangeListener {
14  void propertyChange(PropertyChangeEvent e) { ... }
15  static void main(String[] args) {
16    Point p = new Point();
17    p.addPropertyChangeListener(new Demo());
18    p.setRectangular(5,2); println("p = " + p);
19    p.setX(6); p.setY(3); println("p = " + p);
20  }
21 }
```

Figure 1. Running example, classes `Point` and `Demo`

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

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`. Aspect `BoundPoint`, shown in Figure 2, is used to implement an event firing mechanism by invoking `propertyChange` when a change event is fired. A field `support` and a method `addPropertyChangeListener` are introduced in `Point` by `BoundPoint` at lines 3–7 in Figure 2. Helper class `PropertyChangeSupport` is not shown.

2.1 AspectJ Semantics

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 [5]. There are two categories of shadows: *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.

A *pointcut* selects ("picks out" [18]) 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 and pointcut designator types. A pointcut is dynamic if it is a run-time condition specifier; otherwise, the pointcut is static. A combined pointcut is dynamic if at least one of its component pointcuts is dynamic.

```
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*"); p.reset();}
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(); Point q = p; proceed(q);
28    firePropertyChange(q,"setX",oldX,p.getX());
29  }
30  void around(Point p) : setter(p) { // around2
31    Point p1 = new Point(); proceed(p1);
32    firePropertyChange(p,"setobj",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`

► **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, 18, and 19 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`. ◄

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 [2]. 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` denotes the actual advised call site, and a pair of brackets `[ ]` encloses advices that are invoked by the call to `proceed` in the preceding around-advice. ◄

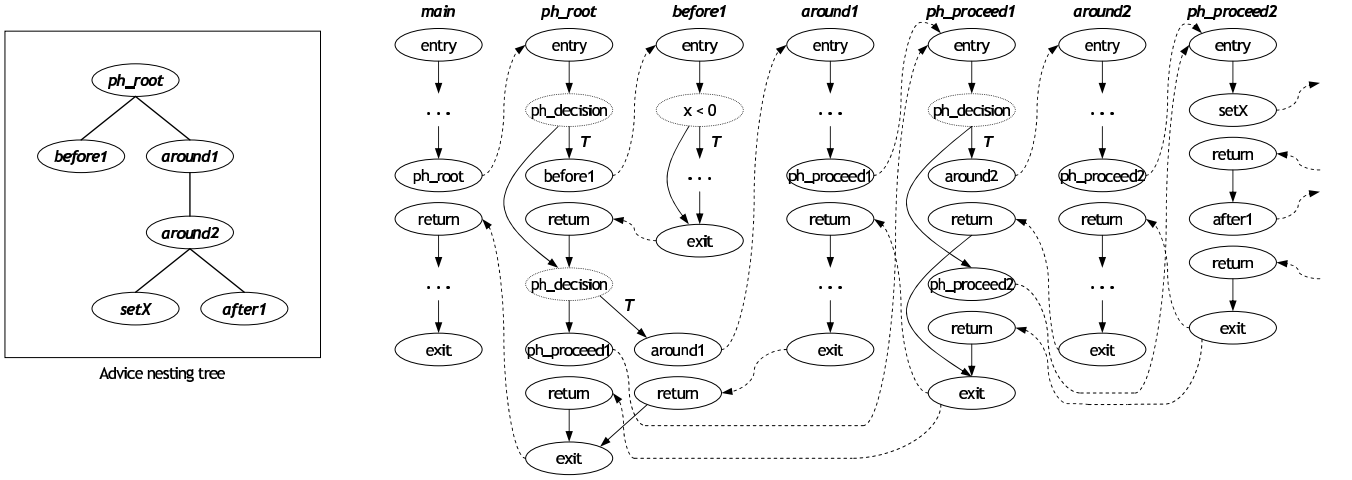

Figure 3. Advice nesting tree and partial interaction graph for shadow `p.setX(6)`

2.2 Control-Flow Representation

In previous work [34] 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.

An ICFG contains (1) standard CFGs that model the control flow within Java classes, within aspects, and 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. The ICFG specifically addresses the situation where multiple advices apply at the same join point, and the existence of dynamic advices. An IG is built for each statement shadow.

► **Example.** Consider shadow `p.setX(6)` at line 19 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 of AspectJ can be used to build a helper data structure, the *advice nesting tree*, which represents the run-time advice nesting relationship. Each tree level contains at most one around-advice, which is the root of all advices in the lower levels of this tree. 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 crosscut call site that could be invoked by the call to `proceed` in A. These advices and the call site appear as if they were nested within A. The advice 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. 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.

3. Data-Flow Representation

In order to apply any dataflow analysis techniques, the key problem becomes: *what data can flow in the interaction graph at a join point, and how can it be represented?* This section describes our technique that builds an ICFG-based data-flow representation by associating data with related ICFG nodes and edges. Our goals are (1) making variables that are defined or used during the interaction explicit for the placeholder methods that will reference them; (2) associating with `ph_decision` nodes the variables that contribute to making run-time decision about invocation of dynamic advices; (3) exposing a minimum set of variables, without introducing any unnecessary variables or extra helper variables in IGs; and (4) keeping a single CFG for each advice declaration, without replicating graphs for different advice applications. Figure 4 shows pseudo-code which is equivalent to the representation for the example from Figures 1, 2, and 3. This representation is the starting point for the interprocedural dataflow analyses described later.

This section describes the handling of call join points and execution join points (see Table 1), which are the most common and useful join point types in AspectJ practice. Due to space limitations, we do not discuss the other three types of join points, although our implementation handles them. In the ICFG, for an execution join point of method `m`, a new method `m$internal` is created, and all CFG nodes and edges from `m` are moved to `m$internal`. A call to `m$internal` is added in `m`, so that the execution join point of `m` is converted to a call join point of `m$internal`. Thus, without loss of generality, the discussion covers only call join points. Our goal is to make explicit the flow of data by creating formal parame-

```
class Demo implements PropertyChangeListener {
 static void main(String[] args) {
   ...
   ph_root1(p,6); // for shadow p.setX(6)
   ...
 }
}
aspect BoundPoint {
 static void ph_root1(Point arg0, int arg1) {
   /* call site for before1 */
   if (...) before1(arg1,arg0);
   /* call sites for around1 */
   if (...) ph_proceed1(arg0,arg1);
   else     around1(arg0,arg1,null,0,0,arg0,0);
 }
 static void around1(Point arg0, int arg1, Point arg2,
                     int arg3, int arg4, Point p, int dv) {
   int oldX = p.getX(); Point q = p;
   switch (dv) {
     /* for shadow p.setX */
     case 0: ph_proceed1(q,arg1);
     /* for shadow p.setRectangular */
     case 1: ph_proceed1_2(q,arg3,arg4);
   }
   firePropertyChange(q,"setX",oldX,p.getX());
 }
 static void ph_proceed1(Point arg0, int arg1) {
   /* call sites for around2 */
   if (...) ph_proceed2(arg0,arg1);
   else     around2(arg0,arg1,null,0,0,arg0,0);
 }
 static void around2(Point arg0, int arg1, Point arg2,
                     int arg3, int arg4, Point p, int dv) {
   Point p1 = new Point();
   switch (dv){
     /* for shadow p.setX */
     case 0: ph_proceed2(p1,arg1);
     /* for shadow p.setRectangular */
     case 1: ph_proceed2_2(p1,arg3,arg4);
   }
   firePropertyChange(p,"setobj",p.getX(),p1.getX());
 }
 static void ph_proceed2(Point arg0, int arg1) {
    /* the original call site */
    arg0.setX(arg1);
    /* call site for after1 */
    if (...) after1(arg0);
 }
 static void before1(int arg0, Point arg1) {...}
 static void after1(Point arg0) {...}
 ...
}
```

Figure 4. Pseudocode for the IG shown in Figure 3.

ters and actual parameters associated with ICFG nodes, in order to enable analysis based on this data-flow representation.

3.1 Declarations for Placeholder Methods and Non-Around-Advices

For a method call join point, the variables that can be referenced are limited to 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 conservatively parameterized with a list of formals $\langle f_0, f_1, f_2, \ldots, f_n \rangle$ that matches the actuals of the shadow call site. Each such method is static and has the same return type as the return type of the method called at the shadow call site; if the shadow does not have/use a return value, the method is declared `void`. For example, consider `ph_root` in Figure 3. The signature of this placeholder method is

```
void ph_root (Point arg0, int arg1)
```

where `arg0` corresponds to the receiver object reference at shadow `p.setX(6)`, and `arg1` corresponds to the actual at this call site.

A non-around-advice that is called by a placeholder method is declared as static with a `void` return type. The formal parameters are its original declared parameters, because a non-around-advice will execute only its own body without affecting the invocation of other advices and the crosscut call site. For example, the signature for advice `before1` in Figure 2 is

```
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` is

```
void afterReturning1(Point arg0,int retval)
```

For an around-advice, the control and data flow are more complicated, because it could invoke other advices and the crosscut method. The handling of around-advices is discussed in detail later.

3.2 Call Sites for Non-Around-Advices

The IG contains call sites that invoke non-around advices; to enable any dataflow analysis, the appropriate actual parameters must be associated with these call site. For an advice that has declared parameters, there have to be one or more data exposer pointcut designators associated with it. There are two kinds of such designators: `target` and `args` (recall Table 1). One can build a formal-to-actual mapping function f_a_map for the parameters in the advice declaration that are specified by `args`. For each such parameter, f_a_map maps its position in the advice's formal list to the position of its corresponding actual in the actual list of the shadow call site. Based on this map, a helper function $fpos$ (short for "formal position") can be defined as follows:

$$fpos(i, pcd) = \begin{cases} 0 & pcd = target \\ f\_a\_map(i) + 1 & pcd = args \wedge cs \text{ is instance} \\ f\_a\_map(i) & pcd = args \wedge cs \text{ is static} \end{cases}$$

where pcd denotes the type of pointcut designator, and cs denotes the shadow call site.

For each before-advice and after-advice, $fpos$ is applied for each parameter p_i in its declaration. As a result, one can obtain the position of the corresponding formal of its caller placeholder method. This formal should be used as the actual for p_i in the created call site. Specifically, for an advice declaration

$$ad(t, p_1, ..., p_n) \ : \ target(t) \ \&\& \ args(p_1, ..., p_n) \ \&\& \ ...$$

a call site of the form

$$ad(F(fpos(0, target)), F(fpos(1, args)), .., F(fpos(n, args)))$$

is created, where $F(j)$ is the j-th formal parameter of the placeholder method that contains the call site.

► **Example.** Consider advice `before1`, called by `ph_root` in Figure 3. Since parameter `x` is specified by the `args` pointcut designator (line 20 in Figure 2), we need to determine its corresponding actual at the shadow call site — in this case, the constant `6`. Since the position of `x` in the parameter list of `before1` is 0, and the position of `6` in the actual list of `p.setX(6)` is 0, the pair $(0, 0)$ is included in f_a_map. Function $fpos$ is then applied for `before1`'s formal parameters `x` and `p`. For `p`, $fpos(1, target) = 0$ and the corresponding formal of `ph_root` is `arg0`. For *x*, $fpos(0, args) = f\_a\_map(0) + 1 = 1$, and the corresponding formal of `ph_root` is `arg1`. Thus, the call site for `before1` inside `ph_root` is of the form `before1(arg1,arg0)`. ◄

The IG also contains a call site for the original shadow. The formals of the caller placeholder method are used as actuals (or receiver) of this call site. If the original shadow is an assignment, a `return$value` local variable in the placeholder method is assigned the return value at the newly created call site. For example, consider the call to `setX` contained in `ph_proceed2` in Figure 3. The created call site is of the form `arg0.setX(arg1)`, where `arg0` and `arg1` are the formals of `ph_proceed2`.

For each after-returning-advice, an additional actual parameter is needed for the last formal parameter. In the caller placeholder method, the `return$value` local should have already been created and assigned the return value of a call to the crosscut method (or to another `ph_proceed` method). This local is used as the last actual at the call site. For example, consider advice `afterReturning1` defined at lines 38–41 in Figure 2. The execution join point for `getX` is converted to a call join point for `getX$internal`, and the shadow becomes the call site for `getX$internal()`. The IG for the shadow contains a call to the crosscut method `getX$internal()`, followed by a call to advice `afterReturning1`. Conceptually, the pseudocode for this IG is as the follows:

```
int ph_root(Point arg0) {
   int return$value = arg0.getX$internal();
   afterReturning1(arg0,return$value);
   return return$value;
}
```

3.3 Handling of Around-Advices

Handling of an around-advice is more complicated because its formal parameters are dependent on the crosscut method and on other advices that are invoked within it. Similarly to a placeholder method, an around-advice must have all necessary parameters of the shadow call site (including the receiver for an instance method), 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 formal parameter list for `ph_proceed2` is `(Point arg0,int arg1)`, `around2` has to take at least these two kinds of formal parameters in order to provide the actuals for the call site, although `around2` itself has only one declared formal parameter (line 26 in Figure 2).

An even more significant problem is that the formal parameters needed by an around-advice could be different for different shadows that the advice matches, because different shadows may call different methods due to the use of wild cards (*) in the pointcut definition. To illustrate, consider again advice `around2` defined in Figure 2. The `setter` pointcut associated with `around2` statically matches five statement shadows. Thus, `around2` can crosscut both `setX` and `setRectangular` calls. For shadows that call `setX`, the formal parameter types needed by `around2` are `(Point,int)`, whereas 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 [1]. However, this violates the fourth goal that was stated at the beginning of this section — the goal 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 dataflow analysis. In fact, our experiments showed that for some benchmarks containing around-advices that match every call site, the size of the program is doubled after it is compiled by the `abc` compiler.

3.3.1 Declarations and Call Sites for Around-Advices

We propose a different approach which does not require the replication of around-advices. For each such advice, our approach considers all shadows that the advice matches, and constructs a globally-valid list $global\_params$ that includes parameters which are required at each shadow. A companion map $spos$ (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 19 and 18 in Figure 1. The actual parameters required at these shadows have types `(Point,int)` and `(Point,int,int)` respectively. Hence, $global\_params$ is `(Point arg0, int arg1, Point arg2, int arg3, int arg4)`. For `p.setX(6)` the starting position of its parameters in $global\_params$ is 0, and for `p.setRectangular(5,2)` the starting position is 2. These positions are encoded in map $spos$. Note that there are three more shadows at which this advice applies; to simplify the discussion, for the rest of this subsection we omit the details related to these three shadows. ◄

List $global\_params$ together with the originally declared parameters of the around-advice are used to build the list of formal parameters for this advice. An additional parameter is also added as the last formal; this parameter is a *decision value* indicating the shadow where the advice is currently applied. For example, the parameter list for `around2` is `(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.

The purpose of including $global\_params$ in the parameter list is to propagate the data from different shadows, in order to use that data to call a `ph_proceed` method, or the crosscut method, for that shadow. Parameter `dv` is used to select among the calls to different `ph_proceed` methods.

In each placeholder method where an around-advice is called, the call site 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 $spos$, (2) the advice's original formal parameters, and (3) the last formal parameter `dv`. For formal parameters corresponding to the shadow, the formals of the caller placeholder method are used as actuals. For the advice's original formal parameters, the actual are constructed similarly to calls to non-around-advices (as described in Section 3.2). A unique shadow ID is used as the actual for formal parameter `dv`. For example, for shadow `p.setX(6)`, the call site for `around2` is

```
around2(arg0,arg1,null,0,0,arg0,0)
```

where `arg0` and `arg1` are the formals of method `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,arg0,1)
```

Note that the last actuals in these two call sites (`0` and `1`) are unique IDs for the corresponding shadows.

3.3.2 Call Sites for Placeholder Methods Inside an Around-Advice

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 the advice could call. The original call to `proceed` in the body of the after-advice is replaced with this group of calls. A special placeholder decision node is created to represent the selection of a placeholder method to be called, and formal parameter `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 (as described in Section 3.3.1) — 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.** Consider the earlier example, where `around2` needs to call `ph_proceed2` at shadow `p.setX(6)`, and to call another placeholder method (e.g., named `ph_proceed2_2`) at shadow `p.setRectangular(5,2)`. The following pseudocode illustrates the control- and data-flow representation:

```
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 call sites 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 formal parameters that are not used by the advice can be cleaned up. For example, after this redundancy removal, the parameter list for `around2` is `(int arg1, int arg3, int arg4, Point p, int dv)`.

3.4 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. This, of course, is essential for subsequent dataflow analysis. There are two kinds of placeholder 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 of node, the associated data is the decision-value formal parameter `dv`. For the second kind, there has to be a run-time condition specifier pointcut associated with the dynamic advice that the node guards. The discussion below considers each pointcut designator that defines a run-time condition, and the data that should be associated with the corresponding placeholder decision node.

As shown in Table 1, there are eight kinds of condition-specifier pointcut designators in AspectJ. For `within` and `withincode`, one can statically determine if the pointcut matches. Hence, they do not contribute to making the run-time decision of whether or not an advice executes. For `target` and `args`, which are also data-exposer designators, the needed data are the parameters that the pointcut specifies. A `this` pointcut designator indicates that the receiver object at the shadow call site must be an instance of the type specified by the pointcut; the data that is needed for the decision is a reference to this receiver object. The `if` pointcut can only reference parameters which are introduced by one or more data-exposer pointcuts. Hence, the data for the decision are again the parameters specified by `target` and/or `args` 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 such pointcuts. Finally, for a composite pointcut, the needed data is the union of the data for its component primitive pointcut designators.

► **Example.** Consider the 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 is the formal of `ph_root` that corresponds to the receiver object — that is, formal parameter `arg0`. ◄

4. Object Effect Analysis

The proposed program representation enables a variety of interprocedural static analyses. Based on this representation, we propose a novel object effect analysis that computes regular expressions to summarize, for each shadow, the field access and method invocation effects of the corresponding interaction graph upon the objects that are passed into the advices from the based code.

► **Example.** For shadow `p.setX(6)` at line 19 in Figure 1, there is only one reference-typed parameter `arg0` passed to its `ph_root` placeholder method. Hence, for this shadow, we generate regular expressions only for `arg0`. Our analysis computes two regular expressions for the corresponding object:

$$(reset|\epsilon)((setX|\epsilon)|(getX((setX|\epsilon)|(setRectangular|\epsilon))))$$

$$((w_x w_y)|\epsilon)(((w_x|\epsilon)|(r_x((w_x|\epsilon)|((w_x w_y)|\epsilon)))))$$

The first expression summarizes the method invocation sequence on the object referred to by `arg0`, and the second one summarizes the field read/write sequence (i.e., the *access path*) on `arg0` (w_{fld} and r_{fld} represent a write effect and a read effect on a field *fld*). These regular expressions encode the paths along which all potential states of the object in this IG can be reached. This information can be used directly to check certain temporal properties, such as typestate-based object protocols (e.g., similarly to [11, 10, 13] for non-aspect-oriented languages). ◄

4.1 Dataflow Problem

At the core of the effect analysis is a context-sensitive, flow-sensitive must-alias analysis that identifies, for each reference-typed formal parameter of a `ph_root` method, a set of variables in the corresponding interaction graph that *must* point to the same object as the formal parameter does. The must-alias information provided by the analysis enables strong updates at assignments and operations via pointers, and therefore can be used to verify many typestate properties that cannot be verified using may-alias information. The regular expressions produced by our analysis may miss effects that could occur on an object of interest. However, using must-occurring events contained in the expressions can eliminate false positives, and thus improve the precision and real-world usefulness of error detection. In fact, similar approaches based on must-alias information have been used in reverse engineering tools [25] and in verification tools [11, 10, 13].

Lattice and transfer functions. We define a lattice of values to state the dataflow problem. Each reference-typed formal f of a `ph_root` method corresponds to a distinct lattice element l_f; this lattice element represents the value of f upon entry to `ph_root`. The lattice also contains a top element $\top$ and a bottom element $\bot$. The lattice for shadow `p.setX(6)` (i.e., for `ph_root1` in Figure 4) is $\{\top, l_{arg0}, \bot\}$. The goal of the analysis is to associate lattice elements with different variables in the corresponding IG.

If $\bot$ is associated with some variable v, this means that v could refer to more than one object, or to an object that is not passed in at `ph_root`. In Figure 4, `p1` in `around2` has this property: $\bot$ is associated with this variable, which shows that `p1` is not included in the must-alias set of `arg0`.

The partial order in the lattice is $\bot \leq l_i \leq \top$, and the meet operation $\wedge$ is defined as follows: $x \wedge \bot = \bot$, $x \wedge \top = x$, $x \wedge x = x$,

and $x \wedge y = \perp$ for $x \neq y$. The meet operation is used by the analysis to merge information about values that are propagated along different execution paths. In particular, consider the last rule. If a variable may refer to one object along one execution path, and to another object along a different execution path, the variable is associated with $\perp$. This resembles constant propagation analysis, which determines expressions that definitely have the same value along all execution paths; similarly, our analysis determines variables that are guaranteed, along all execution paths, to refer to the object that a particular formal parameter referred to at the entry of `ph_root`.

We associate a map $S_n : V \rightarrow L$ with each ICFG node n; here V is the set of variables in the IG, and L is the lattice described earlier. If $S_n(v)$ is some lattice element other than $\top$ and $\perp$, the value of v immediately before the execution of n is guaranteed to be the unique object corresponding to that element. A value $S_n(v) = \perp$ shows that the analysis could not determine that v refers only to a particular object represented by a single lattice element. In the beginning of the analysis $S_n(v) = \top$ for all n and v, indicating that no alias information is currently known.

The effects of ICFG nodes on the solution can be represented by *dataflow transfer functions*. For each node n, the analysis defines a function $f_n : (V \rightarrow L) \rightarrow (V \rightarrow L)$. If S provides information about the values of variables immediately before n, $f_n(S)$ shows the values immediately after n. For any $S : V \rightarrow L$, we will use the notation $S[v \mapsto l]$ to denote a new map that is the same as S except for the value associated with $v \in V$, which is changed to $l \in L$. The key transfer functions are as follows:

- for $v_1 = v_2$: $f_n(S) = S[v_1 \mapsto S(v_2)]$
- for $v_1 = v_2.fld$: $f_n(S) = S[v_1 \mapsto \perp]$
- for $v_1.fld = v_2$: $f_n(S) = S$
- for $v_1 = new\ X$: $f_n(S) = S[v_1 \mapsto \perp]$
- for calls and returns: discussed below
- for other nodes: $f_n(S) = S$

For an assignment $v_1 = v_2$, the analysis propagates the current value of v_2 to v_1. When the value is obtained through an object field in $v_1 = v_2.fld$, a conservative assumption is made that any object reference could be assigned to v_1, and therefore $\perp$ is propagated. More precise treatment of field reads and writes is also implemented in our analysis, by iteratively creating additional lattice elements for field dereferences (e.g., lattice elements $l_{arg0.fld1.fld2}$). Due to space limitations, we do not provide the details of this enhancement. Conceptually, if v_2 can be decided to point to a unique object at a node $v_1 = v_2.fld$, we create a lattice element for $v_2.fld$ and propagate this lattice element to subsequent CFG nodes. The standard k-limiting approach is used in our implementation (with $k = 3$ used for the experimental study) to limit the level of field dereferences to be analyzed.

The transfer function f_p for a path p in the ICFG is the composition of the functions for the nodes and the interprocedural edges on the path. Not all ICFG paths represent possible executions. A *valid* path has interprocedural edges that are properly matched: each (exit, return-site) edge is matched correctly with the last unmatched (call-site, entry) edge on the path, in the sense that both edges correspond to the same call site. The precise solution of the dataflow problem is defined with respect to the set of all valid paths.

The *meet-over-all-valid-paths solution* MVP_n for node n describes the variable values immediately before the execution of n. This solution is defined as

$$MVP_n = \bigwedge_{p \in VP(n)} f_p(MVP_{entry})$$

where $entry$ is the entry node of `ph_root`, and $VP(n)$ is the set of all valid paths p leading from $entry$ to n. For any reference-typed formal f of `ph_root` with a corresponding lattice element l_f, $MVP_{entry}(f) = l_f$. For all other $v \in V$, $MVP_{entry}(v) = \top$ indicating that currently there is no information about v.

4.2 Analysis Algorithm

The dataflow problem presented in the previous subsection is an example of an interprocedural distributive environment (IDE) problem. In IDE problems, the information at a program point is represented by a map from symbols to lattice elements. Sagiv et al. [26] define a general approach for solving such problems precisely. We have instantiated their approach to apply to the problem under consideration. The resulting flow- and context-sensitive algorithm, described in this section, is provably precise in the sense of computing the meet-over-all-valid-paths solution for each node.

Phase 1: Relate formals to variables. The first phase of the analysis computes, for each method reachable from a `ph_root` method, information that relates the values of local variables to the values of the formal parameters of this method. For each node n and local variable v, the analysis computes a set $F_n(v)$ that contains formal parameters of n's method that v *may* be aliased to immediately after n. These sets are propagated along ICFG paths in the method. If at n variable v is assigned values that cannot alias a formal parameter, $F_n(v) = \{\perp\}$. The meet operation is performed at control-flow joint point: if node n does not assign a value to v, and any incoming $F_{n_i}(v)$ contains $\perp$, the resulting $F_n(v)$ must contain only $\perp$; if none of the incoming sets $F_{n_i}(v)$ contains $\perp$, the resulting $F_n(v) = \bigcup_{n_i} F_{n_i}(v)$. Here at node n we have to preserve all possible formal parameters that v may alias at predecessor nodes n_i, because at this time there is no knowledge of the potential aliasing relationships among formal parameters of this method.

If n is a return statement, v is the value that is returned, and $F_n(v)$ does not contain $\perp$, $F_n(v)$ is added to a set SF (short for "summary function") for this method. This set encodes a dataflow summary function in the classical sense of the functional approach for interprocedural analysis by Sharir and Pnueli [27]. In this phase, the analysis considers the strongly-connected-components (SCC) in the call graph and performs a bottom-up processing of the SCC-DAG. If n is a call site of the form $v = o.m(a_1, a_2, \ldots, a_n)$, we consult the sets SF of the possible target methods that could be called by this site, and find a set of actual parameters AP that correspond to the formals contained in these SF sets. For each actual $a_i \in AP$, $F_n(v)$ is updated with set $F_n(a_i)$. Again, if any $F_n(a_i)$ contains $\perp$, the resulting $F_n(v)$ contains only $\perp$.

Phase 2: Propagation of lattice elements. This phase of the algorithm propagates information from callers to callees. Let $S_n(v)$ be the lattice element associated with v immediately before node n. If n is a call node at which v is used as an actual, the value of $S_n(v)$ is propagated to the corresponding formal(s) of the callee method(s). When a method is processed, for each node n and variable v, we replace the formal parameter(s) in $F_n(v)$ with the corresponding lattice element(s). If eventually $F_n(v)$ contains more than one lattice elements, the meet operation is performed on the entire set and the resulting lattice element is used to update $S_n(v)$.

Phase 3: Effect graph building and summary generation. This phase of the analysis performs a depth-first traversal of the ICFG starting from each `ph_root` method, and removes nodes that do not read/write a field or invoke a method on a local variable with which a non-$\perp$ lattice element is associated. The resulting pruned ICFG is an *effect graph*. This graph is essentially a finite state automaton that encodes all reachable states of the objects that flow into this interaction graph. By computing SCCs and bottom-up traversing the SCC-DAGs in an effect graph, we are able to generate regular expressions (as shown in the earlier example) for each incoming object.

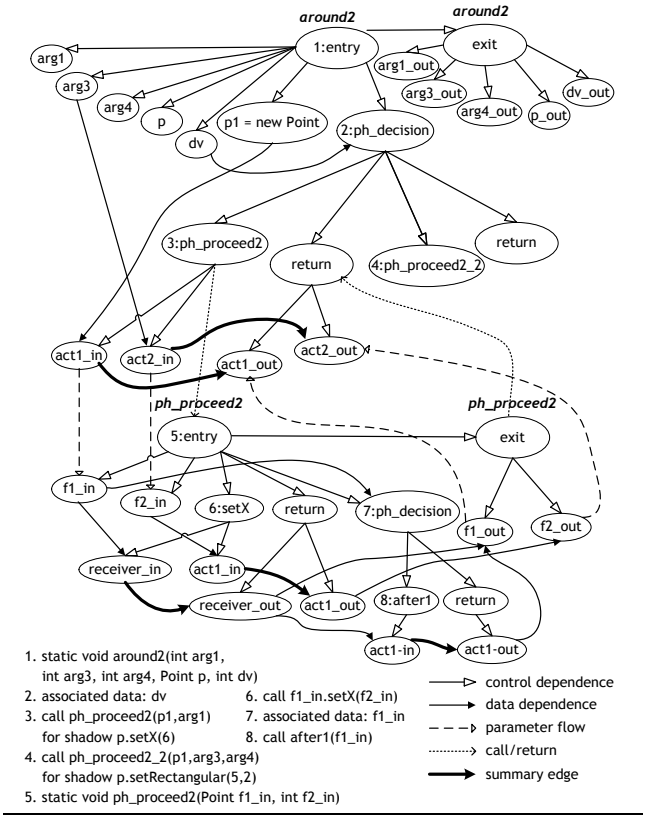

Figure 5. Partial SDG for `around2`

5. Slicing AspectJ Programs

This section outlines a program slicing technique for AspectJ program, as a second proof-of-concept analysis. The dependence analysis used in this technique is another representative of flow- and context-sensitive dataflow analysis algorithms; all such algorithms require the information described in Section 3. Our goal is not to define a complete slicer for AspectJ, but rather to show that the proposed representation contains all necessary information to perform interprocedural dependence analysis. Thus, the analysis is simplified in two ways: (1) it uses a field-based approach that treats all occurrences of the same instance field as aliases, without considering the base object in which the field is contained, and (2) it does not take into consideration the effects of library calls. Call and return edges are determined using class hierarchy analysis. A conservative intraprocedural points-to analysis is used to refine the resolution of virtual calls and to eliminate certain spurious data dependencies.

Building the system dependence graph (SDG). Given the control- and data-flow 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 standard algorithm by Horwitz et al. [16] to build the SDG. A key feature of this algorithm is the computation of *summary edges* that represent transitive dependencies along same-level valid ICFG paths [26]. Such edges are constructed through a bottom-up traversal of the call graph, using the dependence information for a callee to construct the summary edges in a caller.

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).

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

► **Example.** Figure 5 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 advice `after1`. ◄

Slicing AspectJ Software. Standard graph-reachbility-based slicing [16] can be directly applied to the SDG. Each statement in the source code of an AspectJ program, except calls to `proceed`, corresponds to a unique node in the ICFG. Therefore, 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 of which calls a `ph_proceed` method for one shadow. In this case a slice is computed for each call-site node, and the slice union is taken.

6. Experimental Evaluation

To evaluate the proposed techniques, we performed a study which focused on the following research questions:

- What are the ICFG and SDG sizes observed for our approach, compared to using the woven bytecode?
- What is the cost of building the representation?
- How many IG variables are determined to refer to objects flowing to advices, and what is the cost of the effect analysis?
- What is the effect of our approach on slice size and computation time, compared to slicing on the woven bytecode?

Implementation. We have implemented the analyses in our AJANA analysis framework, built on top of the `abc` AspectJ compiler [1]; 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 analyses.

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 [34]; 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 example package. This group of benchmarks has also been used by other researchers [33, 12]. 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. All experiments were performed on a PC with an Intel Xeon 2.8GHz CPU, and run with 512M heap size.

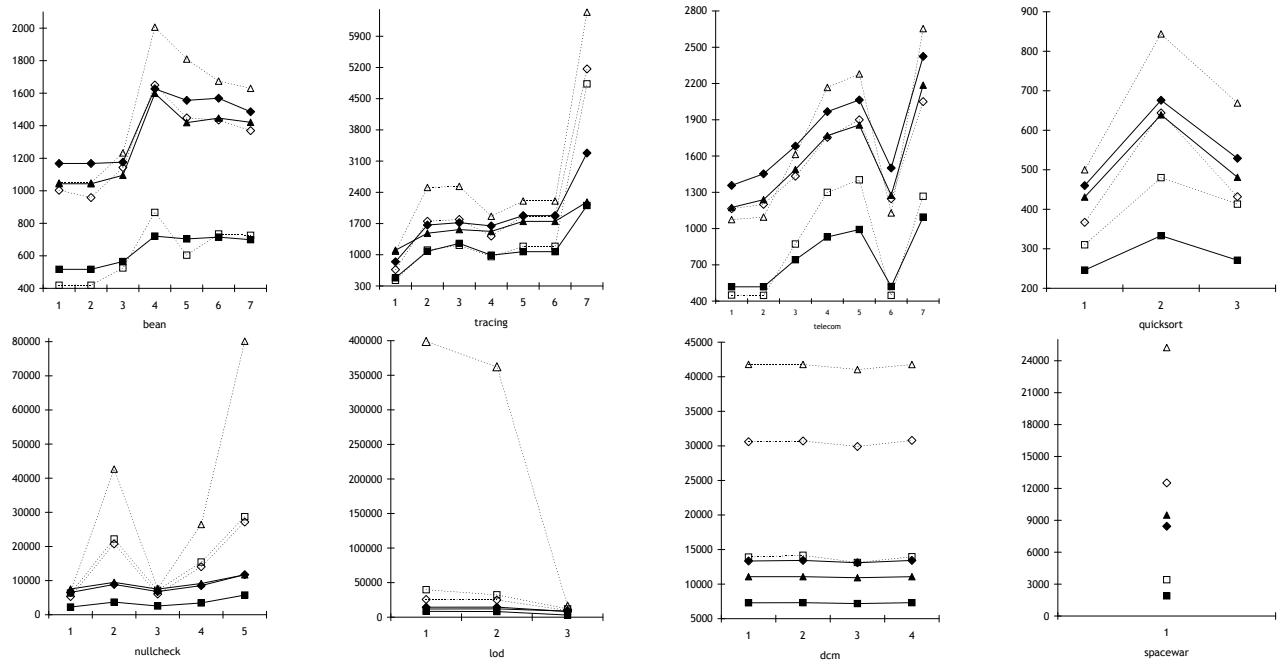

Figure 6. ICFG edges □/■, control-dependence ◇/◆ and data-dependence △ /▲ edges: woven bytecode (□, . . .) and AJANA (■, . . .)

6.1 Study 1: ICFG and SDG Size and Cost

Our first study investigated the sizes of the ICFG and the SDG. Figure 6 compares the graphs constructed by AJANA using the proposed representation and the graphs constructed from the woven bytecode. Nodes in both types of ICFGs corresponded to Jimple statements. The figure shows the number of ICFG edges, the number of SDG control-dependence edges, and the number of SDG data-dependence edges. These results were obtained by running `abc` with advice inlining enabled. We consider edges rather than nodes, because the number of edges is a more important factor that affects the running time of subsequent interprocedural analyses.

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 — in many cases, by at least a factor of two (e.g., for `tracing.7` and all versions of `dcm`).

For most versions, the SDGs built by our technique have slightly more control-dependence edges, due to the artificial decision nodes. 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 interprocedural analysis influenced by data dependencies is likely to incur significant overhead unless it employs our representation.

The algorithm for building the control- and data-flow representation runs in practical time. For example, for programs `nullcheck`, `lod`, `dcm` and `spacewar`, which are the largest among the eight benchmarks, our analysis ran in 156.4, 159.4, 417.7, and 250 ms respectively,[1] whereas the corresponding times for ICFG building from the woven bytecode were 122.0, 122.0, 535.5, and 110 ms. Even though our approach can be a bit slower, these differences are offset by savings in the subsequent analyses.

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. Such independence has critical advantages when relating the analysis results to the original program (e.g., in tools for software understanding and testing), and when considering different weaving compilers or different versions of the same compiler.

6.2 Study 2: Slicing

Our second study investigated the effect of our techniques on the computation of slices for AspectJ programs. 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 are of any relevance for program understanding 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.

In this study we ran the standard slicing algorithm [16] on the SDG built by AJANA and on the SDG built from the woven bytecode. For each benchmark, we choose the versions that 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(a) 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

[1] These times are the averages across all program versions.

Program	Ver	(a)			(b)				(c)
		Size (nodes)	RelRat (%)	Time (sec)	#EG nodes	#EG edges	Precision (%)	Time (sec)	Nodes (%)
bean	v4	367 / 618	98.2 / 54.7	2.2 / 2.6	228	302	100	0.14	56.3
tracing	v7	829 / 1443	98.7 / 77.2	5.8 / 24.2	56	73	100	0.47	35.3
telecom	v7	387 / 230	98.5 / 85.3	2.0 / 1.3	74	96	100	0.19	100
quicksort	v2	309 / 325	98.0 / 44.5	0.3 / 0.8	100	142	100	0.19	75
nullcheck	v2	836 / 5852	97.8 / 37.7	15.9 / 542.7	229	349	100	27.48	1.3
	v5	3313 / 7203	96.1 / 46.0	21.1 / 762.1	388	589	100	154.84	0.8
lod	v1	926 / 3593	97.2 / 64.9	32.0 / 105.1	11	12	92.3	1.63	100
	v2	652 / 3623	97.9 / 60.1	21.2 / 956.7	10	12	91.4	5.36	100
dcm	v2	1444 / 8654	98.2 / 44.3	19.7 / 1011.7	44	54	90	9.91	7.7
spacewar	v1	169 / 1687	97.6 / 68.6	3.9 / 62.8	39	50	76	0.45	15.7

Table 3. (a) Slice size, relevance ratio, and slicing time; (b) Effect graph size, precision, and analysis running time; (c) Number of ICFG nodes in our representation of around-advices, relative to a cloning approach

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 a conservative approximation, we define this set to contain all nodes in compiler-introduced methods. In fact, the relevance ratios shown in Table 3(a) 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 observed cost of SDG construction and slicing.

6.3 Study 3: Effect Analysis

The third study investigated the object effect analysis, using the same program versions. For each `ph_root` method, we ran the effect analysis for each of its reference-typed formal parameters, and generated regular expressions. Table 3(b) shows the results from this experiment. Columns "#EG nodes" and "#EG edges" show the average number, across shadows, of nodes and edges in the effect graph. Column "Precision" shows ratios between the number of EG nodes for our must-alias-based analysis, and the number of EG nodes for an artificial may-alias-based analysis which was defined for the purposes of evaluating the precision of the must-alias information. The may-alias analysis is a modification of the must-alias analysis, and its results provide a conservative overapproximation (i.e., an upper bound) of possible effects. For most programs, the must-alias-based effect analysis achieves perfect precision. The main reason is that an advice body is usually fairly simple— it contains much fewer control flow paths compared to a Java method.

Note that performing such an effect analysis on the woven bytecode is clearly infeasible. For example, inlining of advices into the base code makes an advice a part of the base code. As another example, an around-advice could be broken up into multiple pieces, which are scattered in both the base class and the advice class. Such compiler-specific weaving rules make it extremely difficult for a non-compiler-designer to distinguish between the base code and advices in the woven Java bytecode. Also note that these results were obtained by analyzing the entire interaction graph, rather than a single advice. The analysis of a single advice is not sufficient to provide precise information about its behavior, because its execution is often tied with the execution of other advices.

The running times shown in Table 3(b) include the time used to build the representation together with the execution time of the effect analysis. Clearly, this cost is practical.

Representation of around-advices. Our approach merges the representations of multiple around-advices, as described in Section 3.3. An alternative solution would be to replicate the CFG of an around-advice for each shadow that the advice matches, resulting in a body-cloning-based representation. Table 3(c) shows the ratios between the number of ICFG nodes in our representation of around-advices, and the corresponding number in a cloning representation. For some programs (e.g., `telecom`) our representation does not achieve any reduction, because these programs either do not have around-advices, or an around-advice can apply at only one shadow. For other programs, significant differences can be seen. For example, `nullcheck.5` has only one around-advice, but it can apply at 274 shadows in the program. Cloning the advice body per shadow leads to an extremely large ICFG for this program. Although the proposed representation of around-advices is compact, it can cause propagation along unrealizable paths during a subsequent static analysis. This problem can be solved by using a "shadow-sensitive" analysis that associates a call to `proceed` with a specific shadow that the call invokes, by propagating interprocedurally the constant values of variables `dv`.

Conclusions. For analyzing AspectJ software, especially larger non-toy AspectJ applications, the source-code-based representation proposed in this paper is practical to build, easier to understand, contains significantly fewer nodes and edges, enables pre-weaving analysis of interactions between base code and advices, and can dramatically speed up subsequent dataflow analyses. These results strongly indicate that such a representation could serve as starting point for future work on adapting existing dataflow analyses to AspectJ, and on defining new analyses for AspectJ-specific problems.

7. Related Work

Static analysis of AOP software. The `abc` compiler group [1] developed the AspectBench Compiler for AspectJ, which provides a variety of static analyses and optimizations [3, 4, 6]. Their work focuses on optimizations of the generated bytecode to reduce execution overhead, whereas the focus of our work is representation and optimization at the source-code level abstracting away compiler-specific details, in order to facilitate high-level program analysis and program understanding. We implemented the AJANA framework as an extension to the `abc` compiler, building the ICFG between the static weaving phase and the advice weaving phase.

Rinard et al. [24] present a classification of the interactions between methods and advices. This classification enables develop-

ers to recognize interaction patterns that support modular reasoning and to focus on the causes of potentially non-modular interactions, by employing an existing compositional pointer and escape analysis [32]. However, their work analyzes a single advice, whereas our approach analyzes the entire interaction graph, and therefore achieves higher precision in modeling the behavior of multiple interacting advices. Zhao defines control-flow representations for a variety of testing and analysis tasks for aspect-oriented programs [37, 35, 36, 38]. However, the proposed models do not consider more complex situations such as multiple advices per join point, or dynamic advices. Our previous work proposed a static control-flow model for AspectJ software [34] which serves as the basis for the ICFG used in this paper. However, this approach did not include any data-flow representation that could be used for dataflow analysis.

Interprocedural dataflow analysis. The theoretical foundations for interprocedural dataflow analysis have been investigated extensively (e.g., [27, 22, 26]). Both the object effect analysis and the dependence analysis described earlier are examples of IDE analyses. In the Java community, dataflow analysis has been widely used for compiler optimizations (e.g., [29]), software verification (e.g., [13, 11, 9]), program understanding (e.g., [25]), and error detection (e.g., [21, 8]). As more large-scale aspect-oriented programs are being developed and employed for real-world use, adopting these existing techniques by the aspect-oriented community can benefit numerous compiler construction and software engineering tasks. The work presented in this paper is a step towards applying these techniques to AspectJ software.

There is a large body of work on static slicing [30]. The traditional SDG-based interprocedural slicing algorithm was proposed by Horwitz at al. [16]. Later work addresses the slicing of object-oriented software (e.g., [19]) and of programs with arbitrary control flow (e.g., [28]). Slicing algorithms for aspect-oriented programs were proposed, for example, in [36, 17]. Unlike our work, these efforts do not consider the full complexity of the flow of control and data at join points, or the generality of AspectJ language features. Furthermore, it is not clear how interprocedural dependence analysis would be performed. Our work defines and evaluates experimentally a general program representation which can be used for dependence analysis and for a variety of other interprocedural analyses (e.g., IDE analyses [26]).

8. Conclusions

This paper describes an approach for constructing a static control- and data-flow representation for AspectJ software. This source-code-based technique makes explicit the data exposed during interactions at join points. We use effect analysis, dependence analysis, and slicing as representative client analyses for the proposed approach. Our experiments clearly show that, compared to analysis of the woven bytecode, this representation is better suited as foundation for subsequent static analyses. We also propose a novel effect analysis for understanding and checking of typestate properties of the behavioral effects of multiple interacting advices on incoming objects from the base program. This work creates promising opportunities for future work on adapting many existing Java analyses to AspectJ, and on designing novel AspectJ-specific analyses, for use in various tools for program comprehension, impact analysis, typestate verification, and software testing.

Acknowledgments. We would like to thank the AOSD reviewers for many valuable comments and suggestions.

References

[1] *AspectBench Compiler*. `abc.comlab.ox.ac.uk`.
[2] *AspectJ Compiler*. `www.aspectj.org`.
[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Building the `abc` AspectJ compiler with Polyglot and Soot. Technical Report abc-2004-4, abc Group, Dec. 2004.
[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 *AOSD*, pages 87–98, 2005.
[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI*, pages 117–128, 2005.
[6] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. In *POPL*, pages 11–23, 2007.
[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI*, pages 47–56, 1988.
[8] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, pages 480–491, 2007.
[9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.
[10] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, LNCS 3086, pages 465–490, 2004.
[11] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, pages 12–22, 2004.
[12] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *OOPSLA*, pages 150–169, 2004.
[13] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.
[14] D. Grove and C. Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6):685–746, Nov. 2001.
[15] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.
[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, 1990.
[17] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *ICSM*, pages 178–187, 2004.
[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP*, LNCS 2072, pages 327–353, 2001.
[19] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *ICSE*, pages 495–505, 1996.
[20] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *FSE*, pages 63–72, 2004.
[21] T. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.
[22] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report DIKU-TR94-14, University of Copenhagen, Apr. 1994.
[23] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE*, pages 147–158, 2004.
[24] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *ICSE*, pages 254–263, 2005.
[25] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.
[26] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
[27] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE*, pages 432–441, 1999.
[28] `http://www.sable.mcgill.ca/soot`.
[29] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
[30] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.
[31] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, pages 187–206, 1999.
[32] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD*, pages 190–201, 2006.
[33] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE*, pages 65–74, 2007.
[34] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *International Workshop on Principles of Software Evolution*, pages 108–112, 2002.
[35] J. Zhao. Slicing aspect-oriented software. In *IEEE International Workshop on Program Comprehension*, pages 251–260, 2002.
[36] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *International Computer Software and Applications Conference*, page 188, 2003.
[37] J. Zhao and M. Rinard. System dependence graph construction for aspect-oriented programs. In *MIT-LCS-TR-891*, 2003.