Regression Test Selection for AspectJ Software

Guoqing Xu
Ohio State University

Atanas Rountev
Ohio State University

Abstract

As aspect-oriented software development gains popularity, there is growing interest in using aspects to implement cross-cutting concerns in object-oriented systems. When aspect-oriented features are added to an object-oriented program, or when an existing aspect-oriented program is modified, the new program needs to be regression tested to validate these changes. To reduce the cost of regression testing, a regression-test-selection technique can be used to select only a necessary subset of test cases to rerun. Unfortunately, existing approaches for regression test selection for object-oriented software are not effective in the presence of aspectual information woven into the original code.

This paper proposes a new regression-test-selection technique for AspectJ programs. At the core of our approach is a new control-flow representation for AspectJ software which captures precisely the semantic intricacies of aspect-related interactions. Based on this representation, we develop a novel graph comparison algorithm for test selection. Our experimental evaluation shows that, compared to existing approaches, the proposed technique is capable of achieving significantly more precise test selection.

1 Introduction

As software is modified, during development and maintenance, it is regression tested to provide confidence that the changes did not introduce unexpected problems. Because the size of the regression test suite typically keeps growing, a *regression-test-selection* technique can be employed to reduce the cost of regression testing. A safe regression-test-selection algorithm selects every test case that may reveal a fault in the modified software. Various regression test selection techniques have been developed for procedural languages (e.g., [5, 6, 15, 20]) and for object-oriented languages (e.g., [9, 10, 11, 16, 19, 8, 12].)

Of particular interest for our work is the technique proposed by Harrold et al. [8] for regression test selection for Java based on comparisons of control-flow graphs (CFGs). Given a program P, regression tests are executed to build an edge-coverage matrix which maps each test case to the set of CFG edges exercised by that test case. For a subsequent modified program version P', the CFGs of P and P' are compared to identify "dangerous" edges in the CFG for P. These edges represent program points at which P and P' differ. All and only test cases for P which cover dangerous edges are selected for testing of P'.

Regression test selection for aspect-oriented software. Aspect-oriented software development is a popular approach for modularizing cross-cutting concerns, which simplifies software maintenance and evolution. When aspect-oriented features are added to an object-oriented program, or when an existing aspect-oriented program is modified, the program needs to be regression tested. A precise and safe test selection technique can reduce significantly the cost of regression testing needed to validate the modifications. Aspects can change dramatically the behavior of the original code — for example, without any changes to the original Java code, adding a single AspectJ aspect can arbitrarily change the pre- and post-conditions of many methods. Arguably, regression testing is even more important for aspect-oriented software than for object-oriented software, due to the pervasive effects of small code changes. This paper focuses on one instance of this problem: *what are safe and precise regression-test-selection techniques for systems built with Java and AspectJ?*

The executable code of an AspectJ program, produced by an AspectJ compiler, is pure Java bytecode. Therefore, an obvious approach is to use the regression-test-selection technique from [8] to select tests based solely on the Java bytecode, regardless of whether there are aspects in the source code. However, in addition to the bytecode code that corresponds to the source code (e.g., to bodies of methods and advices), the compiled bytecode of an AspectJ program contains extra code which is inserted by the compiler at certain join points during the weaving process. This *compiler-specific code* checks run-time conditions, decides which advice needs to be invoked, and exposes data from the execution context of join points. Due to this compiler-specific bytecode, the discrepancy between the source code (i.e., Java classes and AspectJ aspects) and the woven Java bytecode can be very significant. This discrepancy creates seri-

ous problems for the graph-based approach from [8], and it will select test cases that in fact do not need to be rerun. In our experimental study, this naive selection approach typically selected 100% of the original test suite for rerunning.

Proposed solution. We propose a new source-code-based control-flow representation of AspectJ programs, referred to as the AspectJ Inter-module Graph (AJIG). An AJIG includes (1) 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 that model the interactions between methods and advices at certain join points. An AJIG captures the semantic intricacies of an AspectJ program without introducing extra nodes and edges to represent the low-level details of compiler-specific code. Thus, AJIGs depend only on the input program and *not* on the implementation details of any particular weaving compiler.

Based on this representation, we define a two-phase graph traversal algorithm that determines a set of dangerous AJIG edges corresponding to semantic source-code-level changes made by a programmer. These edges define the set of test cases that need to be rerun. We implemented the regression-test-selection technique and performed an experimental evaluation of its precision and cost. Our study indicates that the technique has low cost and can effectively reduce the number of test cases selected for rerunning, clearly outperforming the naive approach outlined above.

Contributions. The work described in this paper is the first attempt to systematically address the regression test selection problem for aspect-oriented programs. Our main contributions are as follows:

- A new control-flow representation of the semantics of AspectJ programs, independent of the low-level implementation code introduced by a weaving compiler. This source-code-based representation can serve as the basis not only for regression test selection, but also for various other static analyses for AspectJ.
- A two-phase graph traversal algorithm that identifies differences between two versions of an AspectJ program. The algorithm is specifically designed to take into account the interactions between methods and advices at join points.
- A regression testing framework that implements this technique using the `abc` AspectJ compiler [1].
- An experimental study of the precision and cost of the technique. The results indicate that the AJIG-based approach can effectively and efficiently reduce the number of regression test cases to be rerun.

2 Example and Background

We will use the *bean* program from the AspectJ example package [2] as a running example. We modified the original program by adding several advices to represent more general advice interactions. The example uses aspects to implement an event firing mechanism. Figure 1 shows a class *Point* and a corresponding class *Demo* with a *propertyChange* method which is to be invoked when a property change event is fired. Method *addPropertyChangeListener* and the companion field *support* are introduced in *Point* by aspect *BoundPoint*, shown in Figure 2.

```
class Point {
 int x = 0, y = 0;
 int getX() { return x; } int getY() { return y; }
 void setRectangular(int newX, int newY)
 throws Exception { setX(newX); setY(newY); }
 void setX(int newX) throws Exception { x = newX; }
 void setY(int newY) throws Exception { y = newY; }
 String toString() { println("X="+x+",Y="+y); }
}
class Demo implements PropertyChangeListener {
 void propertyChange(PropertyChangeEvent e) { ... }
 static void main(String[] a) throws Exception {
   Point p1 = new Point();
   p1.addPropertyChangeListener(new Demo());
   p1.setRectangular(5,2); println("p1 = " + p1);
   p1.setX(6); p1.setY(3); println("p1 = " + p1);
 }
}
```

Figure 1. Running example, part 1.

2.1 JIG-Based Regression Test Selection

Harrold et al. [8] present the first regression test selection technique for Java. They define a control-flow representation referred to as Java Interclass Graph (JIG) which extends traditional CFGs. The extensions account for features such as as inheritance, dynamic binding, exceptions, synchronization, and analysis of subsystems.

A JIG contains a CFG for each method that is internal to the set of classes under analysis. Each call site is expanded into a *call node* and a *return node*, and the call node is linked with the entry node of the called method. There is a *path edge* between the call node and the return node to represent the path through the called method. For a method that is external to the analyzed classes, the JIG does not represent the intra-method control flow and a path edge is used to connect the entry node and the exit node of that method. For a virtual call, depending on the run-time type of the receiver object, the call is bound to different methods. Details of the representation of external method calls, exceptions, and synchronization can be found in [8].

After constructing the JIGs of P and P', the technique identifies dangerous edges — edges that may lead to behavioral differences — by performing a synchronous traversal of the JIGs. Given an edge e in P, the algorithm looks for an edge e' in P' which has the same label as e. If e' is found, the target nodes of e and e' are compared to decide

```
aspect BoundPoint {
 /* 'this' is a reference to a Point object */
 PropertyChangeSupport Point.support =
 new PropertyChangeSupport(this);
 void Point.addPropertyChangeListener
 (PropertyChangeListener l) {
   { support.addPropertyChangeListener(l); }
 void firePropertyChange(Point p, String property,
 double oldval, double newval) {
   p.support.firePropertyChange(property,
   new Double(oldval),new Double(newval));
 }
 // ====== pointcuts =======
 pointcut setterX(Point p):
   call(void Point.setX(*)) && target(p);
 pointcut setterXonly(Point p): setterX(p) &&
   !cflow(execution(
    void Point.setRectangular(int,int)));
 // ====== advices ======
 before(Point p, int x) throws InvalidException:
 setterX(p) && args(x) { // before1
   if (x < 0) throw new InvalidException("bad");
 }
 void around(Point p): setterX(p) { // around1
   int oldX = p.getX(); proceed(p);
   firePropertyChange(p,"setX",oldX,p.getX());
 }
 void around(Point p): setterXonly(p) { // around2
   int oldX = p.getX(); proceed(p);
   firePropertyChange(p,"onlysetX",oldX,p.getX());
 }
 before (Point p): setterX(p){ // before2
   println("start setting p.x");
 }
 after(Point p) throwing (Exception ex):
 setterX(p) { // afterThrowing1
   println(ex);
 }
 after(Point p): setterX(p){ // after1
   println("done setting p.x");
 }
}
```

Figure 2. Running example, part 2.

whether e and e' match. If e' is not found, or e' is found but e and e' do not match, e is deemed a dangerous edge. This processing is performed recursively, starting from the main method and from each class initialization method. When testing a program P, the JIG edge coverage of a test suite T is recorded in a coverage matrix, with one row per edge and one column per test case. Testing of P' is done by rerunning test cases from T that exercise dangerous edges.

2.2 AspectJ Semantics

A *join point* in AspectJ is a well-defined point in the execution that can be monitored — e.g., a call to a method, method body execution, etc. 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 [3]. We classify shadows in two categories: *statement shadows* and

```
try{                  around1(){      around2() {
  before1();            if (residue)    before2();
  around1();              around2();    p1.setX(..);
}catch(extype e){      else{          } // around 2
  afterThrowing1();       before2();
  after1(); throw e;      p1.setX(..);
}catch(Throwable e){   }
  after1(); throw e;  } // around1
} after1();
```

Figure 3. Execution of multiple advices.

body shadows. The statement shadow of a method call join point is the corresponding call site. The body shadow of a method execution join point is the body of that method. For example, in Figure 1, call sites *p1.setX()* in *main* and *this.setX()* in *setRectangular* are shadows of the join point of a call to *Point.setX*, and both are statement shadows.

A *pointcut* is a set of join points that optionally exposes some of the values in the execution context. AspectJ defines several primitive pointcut designators; each one is either static (defining a set of join point shadows) or dynamic (defining a runtime condition). For example, static pointcuts are ***call*** and ***execution***, and dynamic pointcuts are ***args***, ***target***, and ***cflow***. A combined pointcut is dynamic if one of its component pointcuts is dynamic; otherwise it is static.

Example. In Figure 2, *setterX* contains all join points where *Point.setX* is called, and *setterXonly* contains only join points where *setX* is called and the call is not within the control flow of an execution of *setRectangular*. Both pointcuts are dynamic because they contain dynamic primitive pointcut designators.

An *advice declaration* consists of an advice kind (before, after, around), a pointcut, and a body of code forming an advice. For an advice associated with a dynamic pointcut, the advice may or may not be invoked at a join point at run time, depending on the evaluation of the corresponding runtime condition. The compiler needs to construct a *dynamic residue* of runtime checks to be performed at the join point to determine whether the pointcut actually matches.

Example. Each of the six advices from Figure 2 could potentially be invoked at call sites *p1.setX()* and *this.setX()* in Figure 1. However, *around2* will not be invoked at the second call site because this call site is within the control flow of an execution of *setRectangular*. At both call sites, dynamic residues will be inserted by the compiler to check if *around2* should actually be invoked at run time.

Whenever multiple advices apply at the same join point, *precedence rules* determine the order in which they execute. If two advices are defined in the same aspect: (1) if either one is an after-advice, then the one that appears later in the aspect has precedence over the one that appears earlier, and (2) otherwise, the one that appears earlier has higher precedence. For brevity, we do not discuss advices from multiple

aspects; they are also handled by our implementation.

Example. The precedence of the advices in *BoundPoint* at either of the shadows is as follows: *before1*, *around1*, *around2*, *before2*, *afterThrowing1*, *after1*. The pseudocode in Figure 3 illustrates the execution semantics of these advices at a join point. We use *residue* as an artificial decision statement to indicate that *around2* may or may not be invoked at run time.[1] If *around2* is invoked, *before2* and the actual call site are invoked by the call to *proceed* in *around2*. Otherwise, *before2* and the call site are invoked by the call to *proceed* in *around1*. If an exception is thrown by the call site or any advice, the control flow goes to *afterThrowing1*. Advice *after1* will be invoked regardless of whether the call site and advices return normally or exceptionally.

```
procedure computeNestingTree
input ads: invocation sequence of advices
output tree: advice nesting tree
create node tree.root
for each advice A in ads in order do
    tree.root.addChild(A)
currAround := none
for each advice A in ads in order do
    if currAround != none do
        if A is AFTER and
            currAround lexically-precedes A do
                continue
        currAround.addChild(A)
        tree.root.removeChild(A)
    if A is AROUND do currAround := A
```

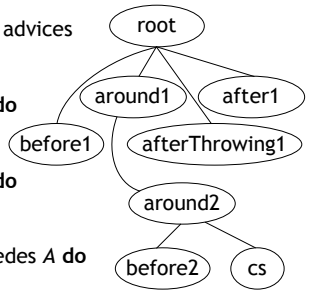

Figure 4. Advice nesting tree.

3 Representation for AspectJ Software

To accurately model AspectJ semantic in a compiler-independent manner, we propose a new control-flow representation, the *AspectJ Inter-module Graph* (AJIG) which extends the Java Interclass Graph from [8] with representations of interactions among methods and advices. The discussion considers only join points corresponding to call site. For a method execution join point (i.e., when the shadow is the body of a method m), a simple transformation can create an artificial method whose body is the same as m, and then replace the body of m with a call to this method. This effectively transforms the join point for m'd body to a call site join point.

Non-advice method calls. An explicit method call can be made in an AspectJ program (1) from a class method[2] to a class method or an aspect method, (2) from an advice to a class method or an aspect method, and (3) from an aspect method to a class method or an aspect method. Because an aspect uses only Java constructs to define non-advice methods, such methods can be treated as class methods defined in special aspect classes. After this adaptation, the JIG representation can be used for all explicit method calls that could occur in AspectJ.

Interactions for advices and methods. The AJIG is designed to represent precisely all interactions involving advices; such interactions are at the core of aspect-oriented programming. Unlike explicit method calls, an advice is invoked implicitly at the shadow of a certain join point. Because the execution of the shadow in the original Java code is completely replaced by the combination of advices that match the shadow, we construct an *interaction graph* (IG) for each shadow to model the semantics of the corresponding join point. For each shadow we replace its CFG nodes and edges in the JIG with the corresponding IG.

When creating an IG, it is essential to consider the situation where multiple advices could be invoked at the shadow and to represent advices with dynamic pointcuts which match the shadow statically, but may or may not match at run time. In the rest of this section we present a technique to build an IG that specifically addresses these two problems.

3.1 Multiple Advice Invocations

Invocation order. Given a set of advices that statically match a method call shadow, their precedence is computed according to the rules described earlier. The call site contained in the shadow is inserted before the first after-advice in the ordered sequence; this call site will be denoted by cs. For example, consider the six advices from Figure 2 which statically match *this.setX()* in *setRectangular*. After taking into account the precedence rules, the resulting sequence is *before1*, *around1*, *around2*, *before2*, *cs*, *afterThrowing1*, *after1*, where *cs* refers to *this.setX()*.

Advice nesting tree. Based on this sequence, we build an *advice nesting tree* which represents the run-time advice nesting relationships. Each tree level contains at most one around-advice, which is the root of all advices in the lower levels of the tree. This construction is illustrated in Figure 4. With each around-advice A the algorithm 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 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 *this.setX()* is shown in Figure 4.

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. For example, *before2* and *this.setX()* at level 4 are invoked by the call to *proceed* in *around2* at level 3, which in turn is invoked by the call to *proceed* in *around1* at level 2. All

[1] Although in this case one can statically decide whether *around2* is invoked, the example shows the semantics of the woven bytecode: the compiler conservatively inserts a residue for a ***cflow*** pointcut.

[2] *Class method* will be used to refer to a method defined in a Java class, and *aspect method* to refer to a non-advice method defined in an aspect.

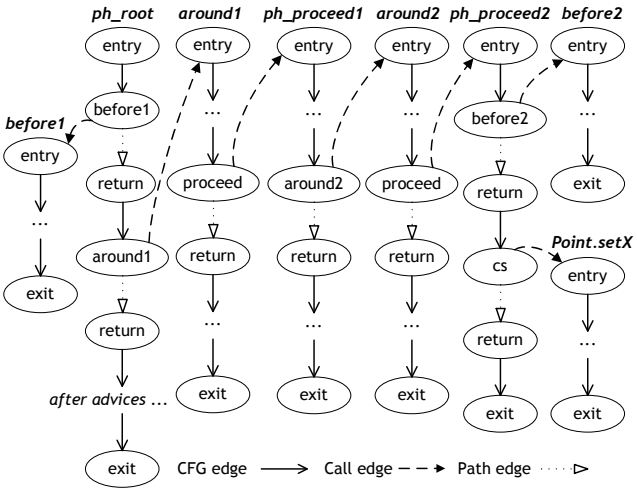

Figure 5. Interaction graph for *this.setX()*.

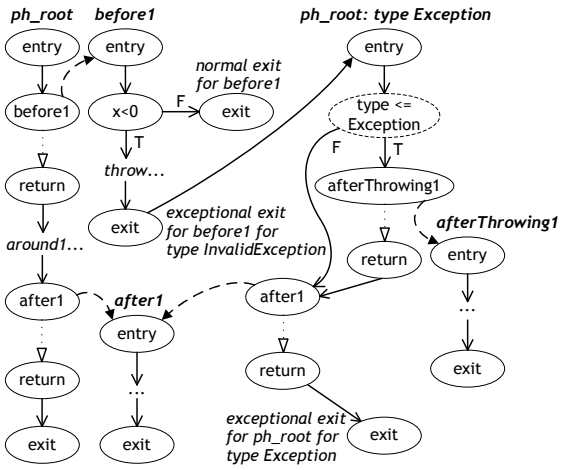

Figure 6. After-advices and exceptions.

advices at level 2 are invoked by the root node, which semantically replaces the execution of the shadow.

Placeholder methods. Clearly, a key question for IG building is how to represent calls to *proceed*. We use a call to a *ph_proceed* placeholder method to represent the call to *proceed* in each around-advice (prefix "ph_" is short for "placeholder"). The placeholder proceed method for an around-advice contains calls to the children nodes in the nesting tree. Such a method is built for each level of the tree in bottom-up manner. The root node of the advice nesting tree corresponds to a special placeholder method *ph_root*. Examples of such methods are shown in Figure 5.

3.2 After-Advices and Exceptions

The execution semantics of after-advices is more complex to model because it is related to the representation of exception handling. After-advices are classified as three types: *afterReturning*, which is invoked when the crosscut method returned normally; *afterThrowing*, which is invoked when the crosscut method threw an exception of the specified type; and *afterAlways*, which is always invoked. Figure 6 illustrates the representation of after-advices in the running example.

Representation of after-advices. Given a sequence of after-advices that appears at some level of the advice nesting tree, sorted in their invocation order, we partition it into two categories. The *normal category* contains advices that are invoked when the crosscut method returns normally, and the *exceptional category* contains the ones that are invoked when the method returns exceptionally. For example, consider *afterThrowing1* and *after1* from Figure 2; both appear at the second level of the tree. The normal category for this level contains *after1* and the exceptional category contains *afterThrowing1* and *after1*.

At each level of the tree, the placeholder method *ph_root* or *ph_proceed* for that level contains a (call,return) node pair for each after-advice. A sub-graph is constructed for the normal category by linking the return node for an advice invocation to the call node invoking the next advice in this category. The sub-graph is added to the end of the placeholder method corresponding to that tree level. In our example, *after1* is called by *ph_root* and its sub-graph appears at the end of the CFG for *ph_root*.

There is also a sub-graph for the exceptional category, with artificial *exceptional entry* and *exceptional exit* nodes. Type *Exception* is associated with both nodes because it is general enough to capture any type of propagated checked exception. For each advice in the exceptional category, a node pair (call,return) is created. If an afterThrowing advice specifies an exception type *ex*, an artificial decision node "$type \leq ex$" is created to show that the run-time type of the propagated exception is a subtype of *ex*. The "true" edge of this decision node is connected to the call node invoking the afterThrowing advice, and the "false" edge is connected to the call node invoking the next advice in the exceptional category (or to another decision node if that next advice is afterThrowing). The decision node indicates that if the run-time type of the exception matches *ex*, then the corresponding afterThrowing advice will be invoked; otherwise, the next advice in this category will be invoked instead.

The nodes in the exceptional sub-graph are linked together by connecting a return node for an invocation of an after-advice with the call node invoking the next advice (or with the corresponding decision node if the next advice is afterThrowing). The exceptional entry node is connected to the call node (or decision node) for the first advice, and the return node of the final advice is connected to the exceptional exit node. The exceptional exit node of the ex-

ceptional sub-graph is also an exceptional exit node of the enclosing placeholder method.

Representation of exception handling. Because the JIG can adequately represent *intra*procedural exception handling within an advice or method, we discuss only the *inter*procedural case. The JIG uses exceptional exit nodes to represent uncaught exceptions within an advice or a method. For each level of the tree that contains after-advices, consider each exceptional exit node *en* in a before-advice, around-advice, and the crosscut method (corresponding to tree node cs) at that level. Node *en* is connected to the exceptional entry node (of the exceptional category) for the CFG of the placeholder method at that level. This means that an exception thrown by a non-after-advice or the crosscut method will be propagated to the exceptional category of after-advices, and then depending on the type of that exception, some after-advices in that category will be invoked.

Each exceptional exit node *en* in an after-advice is connected to the call node invoking the first following after-advice from the exceptional category (or to the corresponding decision node if that next advice is afterThrowing). If the advice is the last one at that level of the tree, *en* is connected to the exceptional exit node. This means that if an after-advice throws an exception, the exception will be propagated to the next afterAlways or afterThrowing advice that could be invoked upon receiving this exception.

Example. Figure 6 shows the representation of exception handling for the running example. Because *before1* throws an *InvalidException*, there is an exceptional exit node in that advice. This node is connected to the exceptional entry node of the exceptional sub-graph in *ph_root*. At run time, the "true" edge of the decision node for *afterThrowing1* will be executed, because *InvalidException* is a subtype of *Exception*. If the *afterThrowing1* advice itself threw an exception, we would link the exceptional exit node in this advice to the call node for *after1* in the exceptional sub-graph, so that *after1* can still be invoked.

3.3 Advices with Dynamic Pointcuts

A dynamic pointcut that statically matches a shadow could potentially *not* match that shadow at run time. Although some types of dynamic pointcuts can be determined to match a shadow at compile time (e.g., *setterX* in *BoundPoint*), in the general case such determination can occur only at run time. We conservatively assume that for all dynamic pointcuts, whether they match a shadow or not has to be determined at run time. Under this assumption, both *setterX* and *setterXonly* are dynamically determined pointcuts. Advices that are associated with dynamic pointcuts will be referred to as *dynamic advices*. All six advices in the running example are dynamic advices.

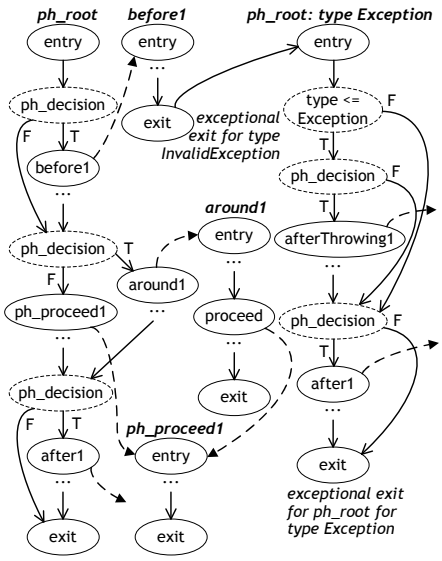

Figure 7. IG with dynamic advices.

For a dynamic advice A, we create a placeholder decision node *ph_decision* for the corresponding pointcut. The "true" edge is connected to the call node for A, meaning that if the decision node evaluates to true, A will be invoked. If A is an around-advice, the "false" edge is connected to a call node for its corresponding *ph_proceed* method — if the pointcut does not match the shadow at run time, the advices that are nested within A will be invoked instead. If A is not an around-advice, the "false" edge is simply connected to the call node for the next advice (or to the exit node) in the graph. All incoming edges of the call node for A are redirected to the decision node for A. Figure 7 shows part of the final interaction graph for *BoundPoint* with the representation of dynamic advices. The graph is semantically equivalent to the pseudocode from Figure 3.

4 Two-Phase Graph Traversal Algorithm

We use the AJIG to define a safe regression-test-selection approach which generalizes the JIG-based approach from [8]. The presence of dynamic advices is problematic for the traversal algorithm from [8]. For example, a test could execute a *ph_decision* node in an AJIG without executing the guarded advice. Because such nodes were created artificially to represent the semantics of dynamic advices, we need to design a new traversal algorithm that appropriately handles decision nodes.

We use the algorithm from [8] to compare edges in Java code. Whenever this algorithm traverses edges e and e', it checks whether their destination nodes N and N' are statement shadows. (Recall that body shadows are converted to statement shadows in the AJIG.) If both nodes are shadows,

their IGs are processed as described below. If neither is a shadow, we proceed with the next pair of edges. If N is a shadow and N' is not, an empty IG for N' is created and IG comparison is performed.[3] If N' is a shadow and N is not, e is added to the dangerous set because it is not known which advices will be invoked if the test is run for P'.

First phase: interprocedural traversal. Given two IGs, the first phase compares the invocation order of advices. This comparison collapses the CFGs of advices and considers only edges in *ph_** methods. The output is a set *DE* ("dangerous edges") of edges in P that are changed in P', as well as a set *FP* ("further processing") of advices whose invocation order remains the same and whose bodies need to be further inspected in the second phase. We start the comparison on the first pairs of edges in *ph_root* and in each exceptional sub-graph which is not reachable from *ph_root*. During the comparison, the logic described below is continuously applied to pairs of corresponding edges.

Four key situations could occur during the comparison. The first case is when e and e' go to call nodes for advices. If the advices are different, e is added to the dangerous set *DE*. Otherwise, the advice is added to set *FP* for further processing; furthermore, if this is an around-advice, the corresponding corresponding *ph_proceed* methods in P and P' are compared recursively. The second case is when e goes to a call node for an advice *ad1*, and e' goes to a placeholder decision node. Here *DE* is updated with e, because the pointcut of *ad1* has changed (from static to dynamic).

The most complex situation is case 3: e goes to a placeholder decision node, and e' goes to a call node for an advice *ad2*. Suppose *ad1* is the advice whose invocation is guarded by the decision node. If *ad1* and *ad2* are the same advice, e is added to *DE* because the pointcut of *ad1* has changed in P'. If they are not the same advice, the comparison cannot stop because if *ad1* is not invoked at run time, the next advice that will be invoked could potentially match *ad2*. The edge labeled "true" is added to *DE* to show that an invocation of *ad1* does not exist in P'. At this point, we need to consider the next advice that will be invoked. This leads to two subcases. First, if *ad1* is a before-advice or an after-advice, the next advice that will be invoked is the one that follows *ad1* in the current placeholder method. Thus, e' should be compared with the edge leaving the return node of the call to *ad1*. Second, if *ad1* is an around-advice, the next advice to be invoked is the first advice in the *ph_proceed* method for *ad1*. In this case we need to compare e' with the first edge in the corresponding placeholder method.

The fourth and final case is when both e and e' go to decision nodes. Suppose *ad1* and *ad2* are the two advices whose invocations are guarded by these decision nodes. A check is performed to determine whether *ad1* and *ad2* have the same signature and the same pointcut. If either the signature or the pointcut is changed, e is added to *DE*. Otherwise, *ad1* is added to *FP*; furthermore, if these are around-advices, their *ph_proceed* methods are traversed recursively to compare all nested advices.

Second Phase: Intraprocedural Comparison. For each advice in set *FP*, we synchronously traverse its CFGs in P and P'. The JIG-based algorithm can directly be applied here to find dangerous edges within the body of the advice.

[3] This is necessary when all advices for N are dynamic, in which case it is possible that a test executes none of these advices at run time.

subject	#Loc	#Versions	#Tests	%mc	%ic
bean	296	8	42	100	100
tracing	1059	6	45	100	100
telecom	870	7	41	100	75
quicksort	111	4	24	100	95
nullcheck	2991	5	63	54.1	76.6
dcm	3423	4	63	54.2	53.8
lod	3075	4	63	54.1	66

Table 1. Subject programs.

5 Empirical Evaluation

Our goal is to investigate empirically the effectiveness and efficiency of the proposed technique. The study considers three research questions:

- What code changes can be introduced by the compiler during weaving, and how do they affect regression test selection?
- How much precision can be gained compared to a technique that operates only at the bytecode level?
- What is the cost of the analysis?

Implementation. We have implemented the AJIG representation and the graph traversal algorithm in a regression testing framework built on top of the `abc` AspectJ compiler [1]. To collect an execution trace, `abc`'s intermediate representation is instrumented after static weaving. The instrumented code is used as input to the advice-weaving component of the compiler. Hence, the generated trace does not include any compiler-inserted information.

Subject programs. Our studies utilize the seven programs shown in Table 1. The first three are included in the AspectJ compiler example package and were used by Xie and Zhao [21]. The remaining four were obtained from the `ajbenchmarks` package [1] used by Dufour et al. [7] to measure the performance of AspectJ programs.

Table 1 shows the number of lines of code in the original program, the number of versions, the size of the test suite, the percentage of methods covered by the test suite (*%mc*), and the percentage of method-advice interactions covered by the test suite (*%ic*). Interaction coverage is defined as follows. A *static interaction* occurs if there is an advice

subject	#Cl1	#Me1	#Loc1	#Cl2	#Me2	#Loc2
bean	3	33	296	4	36	323
tracing	16	111	1059	18	115	2059
telecom	16	102	870	17	105	1110
quicksort	4	18	111	4	20	330
nullcheck	28	196	2991	33	484	5675
dcm	32	226	3423	37	541	6526
lod	32	220	3075	35	290	3855

Table 2. Changes introduced by the compiler.

whose pointcut statically matches a shadow. A *dynamic interaction* occurs if a test executes a static interaction. The interaction coverage is the ratio between the number of dynamic interactions and the number of static interactions.

We originally obtained three versions of *tracing* from the AspectJ web site, and two versions of *quicksort*, three versions of *nullcheck*, two versions of *dcm*, and two versions of *lod* from the `abc` web site. For each program, we created several additional versions to produce complex situations where multiple advices could be invoked at a shadow and where there are both static and dynamic advices. Therefore, rather than changes within bodies of methods or advices, the major differences between versions are the addition or removal of advices, the modification of pointcuts (from static to dynamic or the other way around), and the addition/removal of calls to *proceed* in around-advices. For each program, we made the first version v_1 a pure Java program by removing all aspectual constructs.

To achieve higher interaction coverage, we modified the main methods of the programs to accept user inputs. We developed a test suite for each benchmark to exercise a large number of control-flow paths. There is one test suite for the last three benchmarks, because they are based on the same *Certrevsim* tool for discrete event simulation of certificate revocation schemes.

Threats to validity. As with any empirical study, this study has limitations that must be considered when interpreting its results. We have considered the application of the regression-test-selection techniques to only seven programs, which are smaller than more traditional Java software systems, and we cannot claim that these results necessarily generalize to other programs. However, the first four programs we chose are known examples which have been used in the evaluation of previous work [14, 21, 7], and the last three are among the largest in the `ajbenchmarks` suite that we could find as real-world AspectJ applications.

Threats to internal validity mostly lie with possible errors in our implementation and measurement tools that could affect outcomes. To reduce these threats, we performed several sanity checks. We also spot checked, for many of the changes considered, that the results produced by the two phases of our approach were correct.

Compiler-introduced changes. Our first goal was to understand the changes that the compiler could introduce during weaving, and their effects on regression test selection. We compiled the benchmarks into bytecode using the `abc` compiler, which currently outperforms the `ajc` compiler [3]. We then used the Dava decompiler included in the Soot framework [18] to decompile the woven code back to Java source code to inspect the changes. Table 2 provides an overview of changes introduced by the compiler for the original AspectJ program versions v_2. Columns *Cl1*, *Me1*, *Loc1* and *Cl2*, *Me2*, *Loc2* illustrate the numbers of classes, methods, and lines of code before weaving and after weaving. (Inaccuracy for #Loc could be introduced by Dava.)

Clearly, significant amount of additional code is inserted by the weaving compiler. For *tracing*, *nullcheck*, *dcm* and *lod*, the number of methods (and lines of code) grows dramatically. We found that the advices defined in these programs can crosscut almost every method in the base classes. Some of these advices are around-advices, which makes the compiler generate hundreds of *around_** methods, each of which is called to replace a shadow statement. We observed various examples of changes that actually had nothing to do with the modification between versions:

- In front of an existing advice in *tracing*, we added a new one that matches a different shadow. As a result, the compiler changed the name of the existing advice as well as the call site that invokes this advice.
- When we removed a control-flow path in a before-advice in *bean*, this advice method disappeared. Instead of calling the advice method, the compiler inlined the whole advice body at the shadow.
- When the before-advice was inlined, the names of some local variables in the method that contains the shadow were changed because of name conflicts.
- When we added an afterThrowing advice in *bean*, the compiler generated a try-catch block that enclosed the shadow, regardless of whether the advices and the call site would actually throw an exception.
- When the pointcut *setterXonly* was added in a version in *bean*, dynamic residue was inserted at both of shadows *this.setX* and *p.setX*. However, the pointcut would never match the former.

The key observation is that such changes to the woven bytecode are due to low-level details of the compiler implementation, but they are *not* program changes that should affect regression test selection. Unfortunately, such changes can prevent the existing JIG-based algorithm [8] from selecting only the changes made in the source code.

Study of two regression-test-selection techniques. We performed a study of two regression-test-selection techniques: the existing JIG-based technique from [8] and the

b2	100/57.1		b3	100/52.3	76.2/76.2
b4	100/57.1	71.4/71.4	b5	100/90.0	100/85.7
b6	100/90.0	100/85.7	b7	100/90.0	100/85.7
b8	100/90.0	100/85.7	tr2	100/95	
tr3	100/95	100/95	tr4	100/95	100/50
tr5	100/95	100/95	tr6	100/95	100/50
te2	71.4/39.3		te3	100/53.6	97.8/57.1
te4	100/71.4	100/71.4	te5	100/71.4	100/28.6
te6	100/71.4	100/28.6	te7	100/28.6	100/39.3
q2	100/100		q3	100/100	100/0
q4	100/100	100.95.2	n2	100/90	
n3	100/98.4	100/90	n4	100/90	100/90
n5	100/98.4	100/90	n6	100/98.4	100/48.2
d2	100/98.4		d3	100/98.4	100/90
d4	100/98.4	100/100	d5	100/90	98.4/0
l2	100/98.4		l3	100/98.4	100/90
l4	100/90	100/90	l5	100/98.4	100/90

Table 3. Regression test suite reduction.

proposed AJIG-based technique. Although there are variations of JIG-based selection [12], we consider only edge-level selection because our approach identifies changes at the edge level. The study evaluated two cases: (1) P is a Java program and P' is an AspectJ program, and (2) both P and P' are AspectJ programs. Hence, for each program, we compared each version to its pure Java version v_1 and to its original AspectJ version v_2.

Table 3 shows the test suite reduction achieved by the two techniques. The numbers show the *percentage of test cases that need to be rerun*. Each AspectJ program version is labeled with its number — e.g., *b3* corresponds to version v_3 of *bean*, *te5* is version v_5 of *telecom*, etc. There are two table cells for each AspectJ program version v_i ($i \geq 2$). The first cell considers P to be the pure Java version v_1 and P' to be v_i. The second cell considers P to be the original AspectJ version v_2 and P' to be v_i ($i \geq 3$). In each cell of the table, a slash "/" separates the percentage of test cases selected by the edge-level JIG-based technique, and the percentage of test cases selected by our approach.

In most cases, our technique outperforms the JIG-based technique. For some cases such as the second cell for *q3*, our technique selected no test cases, whereas the JIG-based technique selected all of them. In this particular case there was a removal of a dynamic after-advice that statically matches every call site, but is never executed at run time. The dynamic residue inserted by the compiler at each call site forced the JIG-based technique to select all test cases. As another example, no tests could be avoided by our approach for the first cells of *q2*, *q3*, and *q4*, because these versions contain advices that crosscut base Java methods which are always executed at run time (e.g., `main`). Hence, it is impossible to avoid any tests when each of these versions is compared with the pure Java version v_1.

Analysis cost. The analysis runs in practical time, compared to the compilation time. For example, for the three largest programs *nullcheck*, *lod* and *dcm*, our analysis ran in 1.88, 0.78 and 0.94 seconds as part of the compilation, while the entire compilation process finished in 11.50, 11.39 and 12.36 seconds respectively.

6 Related Work

The `abc` compiler group [1] developed the AspectBench Compiler for AspectJ, which provides a variety of static analyses and optimizations [3, 4]. We implemented our technique as an extension to the compiler, building the AJIG representation before advice weaving.

Many researchers have considered the problem of regression test selection (e.g., [6, 15, 5, 8, 15, 19, 13]). The closest related work is the JIG-based approach by Harrold et al. [8]. As evident in our experimental study, this approach does not appear to be effective for woven bytecode, and typically selects 100% of the test cases. Previous work by Xu [23] and Zhao et al. [27] proposed approaches for selecting regression test for aspect-oriented programs. Both of these approaches suggest to use a "clean" CFG to model the control flow of aspect-oriented programs, excluding the compiler-specific information. However, [23] leaves the precise definition and evaluation of the "clean" CFG for future work. The CFG model proposed by [27] does not consider the situation where multiple advices apply at a shadow, or the existence of dynamic advices. This work does not implement or evaluate the proposed approaches.

Rinard et al. [14] present a classification of the interactions between methods and advices, and apply a pointer and escape analysis. Zhao defines control-flow representations for a variety of testing and analysis tasks for aspect-oriented programs [26, 24, 25]. The proposed models are fairly lightweight: they do not include representations for any complex situations, such as multiple advices or dynamic advices. Furthermore, there are no implementations or evaluations of these models.

Souter et al. [17] develop a test selection technique based on concerns. To reduce the cost of running tests, they propose to instrument only the concerns of interest. They also propose to select or prioritize tests for the selected concerns. Xu and Xu [22] present a specification-based testing approach for aspect-oriented programs. They create aspectual state models using flattened regular expressions. Xie and Zhao [21] describe a wrapper class synthesis technique and a framework for generating test inputs for AspectJ programs. These efforts focus on testing of aspect-related features, whereas our work is the first attempt to systematically perform regression test selection for general AspectJ code.

7 Conclusions

We propose a regression-test-selection technique specifically aimed at complex AspectJ language features. The approach builds the AJIG control-flow representation, at the core of which are graphs encoding the interactions among multiple advices. Because the AJIG is built from the source code, it does not represent any compiler-specific code. Thus, when the AJIGs for two program versions are compared, the identification of dangerous edges captures the semantic differences between the two versions while abstracting away the low-level details that are specific to a compiler implementation. We propose a graph traversal algorithm in which the key step is to compare the calling structure of interaction graphs. Our experimental study indicates that (1) low-level compiler-specific changes occur commonly, and (2) our technique can effectively reduce test suite size, clearly outperforming the JIG-based approach.

The AJIG is a general control-flow model for AspectJ software which could serve as basis for various static analyses for AspectJ: regression test selection [23, 27], change impact analysis [24], data flow analysis [26], program slicing [25], analyses for program understanding, etc. In the future we will investigate such uses of this representation.

Acknowledgments. We would like to thank the ICSE reviewers for their 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. Optimising AspectJ. In *Conf. Programming Language Design and Implementation*, pages 117–128, 2005.

[4] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. In *Symp. Principles of Programming Languages*, pages 11–23, 2007.

[5] T. Ball. On the limit of control flow analysis for regression test selection. In *Int. Symp. Software Testing and Analysis*, pages 134–142, 1998.

[6] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: A system for selective regression testing. In *Int. Conf. Software Engineering*, pages 211–220, 1994.

[7] 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*, pages 150–169, 2004.

[8] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 312–326, 2001.

[9] P. Hsia, X. Li, D. C. Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *J. Software Maintenance*, 9(4):217–233, 1997.

[10] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. Firewall regression testing and software maintenance of object-oriented systems. *J. Object-Oriented Programming*, 1994.

[11] D. C. Kung, J. Gao, P. Hsia, F. Wen, and Y. Toyoshima. Change impact identification in object oriented software maintenance. In *Int. Conf. Software Maintenance*, pages 202–211, 1994.

[12] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Symp. Foundations of Software Engineering*, pages 241–251, 2004.

[13] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 432–448, 2004.

[14] 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.

[15] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Software Engineering and Methodology*, 6(2):173–210, 1997.

[16] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *J. Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[17] A. Souter, D. Shepherd, and L. Pollock. Testing with respect to concerns. In *Int. Conf. Software Maintenance*, pages 54–63, 2003.

[18] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Int. Conf. Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[19] L. J. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. In *10th Annual Software Quality Week*, May 1997.

[20] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Int. Conf. Software Maintenance*, pages 262–270, 1992.

[21] 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.

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

[23] G. Xu. A regression tests selection technique for aspect-oriented programs. In *Workshop on Testing Aspect-Oriented Programs*, pages 15–20, 2006.

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

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

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

[27] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ programs. In *Workshop on Testing Aspect-Oriented Programs*, pages 21–26, 2006.