On bytecode slicing and AspectJ interferences

Antonio Castaldo D'Ursi
DEI, Politecnico di Milano
Via Ponzio 32
Milano, Italy
ercasta@gmail.com

Luca Cavallaro
DEI, Politecnico di Milano
Via Ponzio 32
Milano, Italy
cavallaro@elet.polimi.it

Mattia Monga
DICo
Universita degli Studi di Milano
Via Comelico 39
20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

AspectJ aims at managing tangled concerns in Java systems. Crosscutting aspect definitions are woven into the Java bytecode at compile-time. Whether the better modularization introduced by aspects is real or just apparent remains unclear. While aspect separation may be useful to focus the programmer's attention on a specific concern, the oblivious nature of the weaving makes it difficult to figure out the behavior of the whole system. In particular, it is not easy to figure out if two aspects interfere one with the other. We built a bytecode slicer called XCutter in order to study which part of the woven code is affected by the application of an aspect. However, our experiments show that a static analysis of AspectJ woven bytecode does not give the expected results, unless the code is properly annotated.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Language Constructs and Features**]:

General Terms

Languages, Verification

Keywords

Program analysis, Slicing, aspect-oriented programming, AspectJ, interference analysis

1. INTRODUCTION

AspectJ [1] is the most successful language embodying the idea of aspect-oriented programming, introduced by Kiczales et al. in [2]. In AspectJ, crosscutting entities called *aspects* are *woven* into traditional object-oriented (Java) bytecode at compile-time. Nevertheless, events that can trigger the execution of aspect-oriented code are run-time events: method calls, exception handling, and other specific points in the control flow of a program. The basic idea is that aspects describe crosscutting computations (pieces of *advice*) by referring to an abstract view of the system and composition is performed by the automatic weaving process, which produces a standard Java bytecode application.

In principle, the various aspects should not interfere with one another, and they should not interfere with the evolution of classes. Currently, non interference in presence of class evolution is really hard, and programmers should be very careful in writing aspects that make use of the implementation details of classes as little as possible if they want to be able to reuse their aspects. Moreover, it is still not clear how to cope with the difficult problem of aspect interaction. In fact, the code affected by an aspect is oblivious about that, i.e., its text does not contain any clue about which aspects might or will be advised on it. Thus, by looking at a given statement, programmers may have a hard time figuring out if one of the aspects of the system will influence it. We believe that this represents a limit in the current AspectJ approach, since it means that the actual separation of aspects is in a sense only apparent. In other words, while aspect code units are physically separated, one has always to keep all of them in mind while coding the other parts of the system, since every aspect could potentially influence any other component.

In order to assess the complexity of the weaving in an AspectJ program, we proposed[3, 4] to use program analysis techniques to measure how large is the portion of a program potentially (i.e., statically known) affected by an aspect. We suggested this could be used to study aspect interactions. In fact, roughly speaking, if the portions affected by two aspects do not overlap, this is a sufficient condition to state that they do not interfere. More precisely, let a *code unit* be an aspect or a class of a system. We say that an aspect A does not interfere with a code unit C if and only if every interesting predicate on the state manipulated by C is not changed by the application of A. For example, if an object x manipulated by C exists such that the predicate $x \leq 0$ must hold for the correctness of the system, A does not interfere with C only if C woven with A preserves $x \leq 0$. This definition captures only interferences caused by inconsistencies in the state manipulated by the code units: other types of clashing are not considered.

This can be used to derive an operational test to find out aspect potential collisions [3]. If A_1 and A_2 are two aspects and S_1 and S_2 the corresponding backward slices [5] obtained by using all the statements defined in A_1 and A_2 as the slicing

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.

criterion, A_1 does not interfere with A_2 if $A_1 \cap S_2 = \emptyset$. In fact, the set S_2 contains all the statements that affect the slicing criterion (a set containing all the statements of A_2).

To this end, we built XCutter [1] (to be pronounced *crosscutter*) a tool to do backward static slicing on Java bytecode and to map bytecode entities back to the AspectJ code. In this paper we report about the challenges we encountered in building such a tool and some preliminary results about its use. The paper is organized as follows: Section 2 briefly surveys slicing techniques and talks about our work to build a working slicer, Section 3 talks about the preliminary results we obtained using our tool on AspectJ programs for interference detection, and Section 4 finally discuss the lessons we learnt.

2. SLICING OBJECT AND ASPECT ORIENTED PROGRAMS

Program slicing is a program analysis technique introduced by Weiser in the '80s [5]. A backward slice is a set of program instructions that, fixed one or more instructions as a *slicing criterion*, influence the criterion, i.e., change the right-values of the statements included in the criterion. Ottenstein and Ottenstein proposed a slicing technique based on a graph representation of the program in [6], in which directed edges represent data and control dependencies between instructions. The original proposal was about a technique to analyze a single procedure of a procedural program. In [7] the graph-based approach was extended to handle programs including interacting procedures calls, and an algorithm that could correctly take the calling context into account was proposed. This algorithm performs two phases of reachability analysis, which consider different kinds of edges, to preserve the calling context. The resulting slice is a set of graph nodes that is mapped back onto the program source. Slicing techniques were further studied and applied to object oriented programs by Liang and Harrolds in [8].

Object oriented slicing techniques, unfortunately, cannot be used as is to analyze aspect oriented programs, since the weaving introduces data dependencies that have to be taken into account. A graph representation for the AspectJ language was proposed by Zhao in [9]. His work relies on the graph representation presented for object oriented programs and adds representation for some of the constructs of AspectJ. Pieces of advice are represented as methods, pointcuts are represented adding a *pointcut edge* from the entry of a piece of advice representation to the point in the base system code captured by the pointcut. Intertype declarations are represented adding the representation for the field or method introduced and binding it with an *introduction edge* to the interested point of the base system. However, the dynamic nature of pointcut definitions (that can even apply to pieces of advice to which are attached) is neglected.

Slicing AspectJ programs by considering aspects as first class entities, is appealing, since it allows for not considering the actual implementation of the weaver. However, in order to build working tools one has to deal with the details of the expressive power of all AspectJ constructs. This effort actually replicates the weaving task. Therefore, in [4] we proposed to exploit the fact that AspectJ programs are eventually translated into Java bytecode, and the latter is an object oriented language, to which the mainstream state of the art of program analysis can be applied.

Our strategy is divided in four steps:

1. Compile Java classes and aspects using an AspectJ compiler and weave aspects into an executable program.
2. Apply existing slicing algorithms to the resulting bytecode.
3. Obtain a slice as a set of bytecode statements.
4. Map these statements back onto the original source code of the program.

Working at bytecode level may seem inappropriate, since in bytecode there is no distinction between the aspect oriented and the object oriented parts of an AspectJ program. In fact, the weaving process translates aspects into classes, pieces of advice in methods and pointcuts into method invocations. Thanks to this, the mapping of bytecode instructions onto source code can be done rather efficiently and precisely. Some problems about intertype declarations remain (see Section 2.3.5): these could be in future resolved by a suitable use of bytecode annotations by the AspectJ compiler. XCutter, our backward static slicer, can analyze both AspectJ and Java programs, since it works at bytecode level.
A tool similar to ours is Indus [10], a slicer for Java that works at the bytecode level: unfortunately it was made available when our effort was already begun and initially released with a license [11] incompatible with our commitment to produce an open source product. Indus can slice multithreaded programs, while our current prototype can not. It is based on a context-sensitive points-to analysis, but context-sensitivity is not fully exploited by its slicing algorithm. For example, the context-sensitive points-to analysis can distinguish between different instances of internal data structures of different `Vectors`, but the slicing algorithm does not distinguish between modifications to two different `Vectors`, thus obtaining the same precision level resulting from the use of a context-insensitive points-to analysis. Moreover, Indus is focused on slicing Java programs. AspectJ-specific slicing strategies could not be implemented on top of it, so it could not be used for interference analysis.

2.1 Slicer Architecture

XCutter is built on top of Soot, a program analysis framework for Java [12]. Soot provides an intermediate representation of Java bytecode called Jimple– a three-address typed representation suitable for analysis– and supports intraprocedural analyses. Moreover it is accompanied by Spark, a framework for points-to analysis.

XCutter is structured as a series of analyses which run inside the Soot framework. As depicted in Figure 1, the analysis starts by compiling the AspectJ program sources and

[1]The source code of the tool is available at `http://www.elet.polimi.it/upload/cavallaro/thesis/Xcutter.jar` under the terms of a GPL license.

weaving the resulting bytecode. The latter is then imported in Soot and translated into Jimple. This intermediate representation undergoes a series of preliminary analyses, used to discover some information necessary to build a slice, and is used as input for the slicing algorithm, described in Section 2.3. The resulting slice is finally mapped back onto the source code of the AspectJ program.

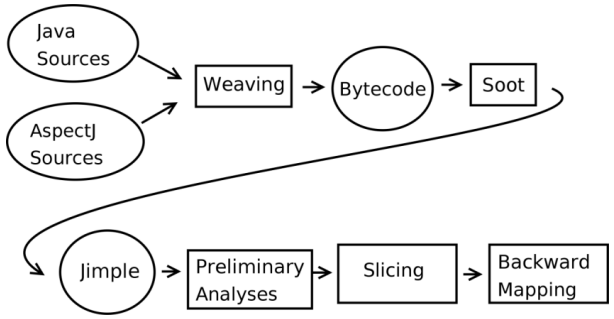

Figure 1: Architecture of XCutter

2.2 Preliminary analyses

Preliminary analyses compute data and control dependencies between instructions of the program. Instructions are annotated with information on the discovered dependencies, which are used by the slicing algorithm to compute the backward static slice.

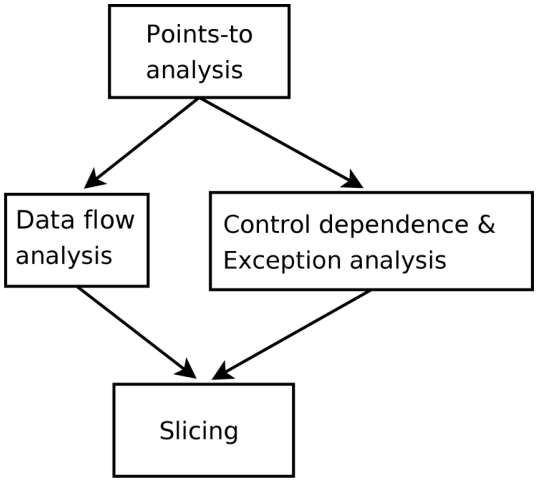

Figure 2: Overview of analysis order

The preliminary analysis starts computing the points-to information by using the Spark framework [13]. This analysis computes the set of Java objects that a given variable in the program may point to. These results are then used by both data flow and control dependence and exception analysis (see Figure 2).

The data flow analysis computes the reaching definitions in the program. For efficiency reasons we divided this analysis in local data flow analysis and reference data flow analysis. Local data flow analysis computes reaching definitions between Jimple local variables contained in a single method. For this analysis we adapted the algorithm described in [14]. Reference data flow analysis computes data dependencies caused by definitions and uses of static fields, object fields, and arrays. This analysis needs a side effects analysis as a preliminary step. The side effect analysis computes which object fields, arrays and static fields may be used or modified by each method in the program. Each method is, initially, analyzed by itself, then the information found for a method is propagated through the call graph. Reference analysis uses the results of the side effect analysis to model the effect of method call statements. Using this strategy the reference analysis can be performed intraprocedurally, improving efficiency.

The control dependence analysis aims at finding intraprocedural control dependencies. The algorithm used was adapted from [14]. The exception analysis was adapted from the one proposed in [15]. Exceptions may introduce intraprocedural and interprocedural control dependencies in a program. Methods in the program are searched, from call graph leaves to the main method, to find if they might throw exceptions. If an exception is thrown there is a control dependence from the throwing instruction to all the following instructions, in the method body. Moreover the method is searched for an appropriate catch clause. If it is found, it means there is a control dependence from the throwing instruction to the instructions contained in the catch block. If no catch block is found the information about the thrown exception is propagated to the callers. If the callers contain an appropriate catch block there is an interprocedural control dependence from the thrower instruction to the instructions of the catch block, else the information is further propagated to the callers.

The results of data flow and control dependence and exceptions analyses are annotated in tags associated with Jimple instructions and methods, and are used as input for the slicing algorithm.

2.3 The slicing algorithm

Existing slicing algorithms for procedural and object oriented programs ([7], [8], [16]) require the construction of a graph representing the analyzed program. Most features of the Java language have been separately taken into account, and algorithms to create corresponding graphs have been proposed. However, creating a graph that correctly takes into account the whole Java language requires theoretical work to merge different approaches. Our slicing algorithm is not graph-based. Instead of building a graph representing the entire program, slicing is performed using results of preliminary analyses, which compute dependencies between statements. This makes the slicing algorithm more complex than a graph-based algorithm, because dependencies between instructions are not represented explicitly by edges. However, several features such as exception handling and polymorphism are easier to manage without an unnecessary pollution of ad-hoc edges. In the following, we describe the engineering challenges we encountered in building a working tool: most of problems we faced are extensively studied in the program analysis literature. Notwithstanding that, putting together independent results in an effective prototype was a hard work, mostly absent in the publicly available code.

2.3.1 Library and application methods

Any non trivial Java program uses some library method. This means that a lot of library methods are part of the program, because they are transitively called by application code. Traditional algorithms require a detailed analysis of the entire program and the creation of a graph for every library and application method. While experimenting on small AspectJ examples, we noticed that usually a large part of the program is composed by library methods. Our interference analysis deals with slices containing instructions belonging to aspects, which are not contained in the Java library. So we decided to keep the distinction between library and application methods, and analyze them with different precision levels. In particular, detailed data and control dependence analysis is performed on application methods only. The slicer analyzes library methods to detect which values are used or defined by these methods, but it does not compute dependencies between instructions of a library method. This also affects how method calls are handled. Information on side effects and thrown exceptions is necessary to take into account data and control dependencies. However, calls to library methods are treated atomically. In fact, when an instruction calling a library method is put into the slice, the whole library method, and the method it transitively calls, are considered to be part of the slice. The difference between library methods and application methods is decided by the user, by choosing which packages contain application methods. Moreover, the analysis can be configured to treat some library methods with the same level of detail used for application methods, using a *depth* parameter. The increased precision is used for library methods whose distance from application methods in the call graph is smaller than *depth*.

2.3.2 Using dependencies

The slicing algorithm uses dependencies computed by preliminary analyses to add instructions to the slice. Although the slicing algorithm is not graph based, we called node the entity used to represent instructions. However, new nodes are created only when new instructions are added to the slice. The algorithm uses several kinds of nodes. When a new instruction is added to the slice, a new node is created, whose kind depends on the included instruction. Table 1 summarizes node types and the corresponding actions, and Figure 3 shows how the algorithm decides the type of a node when a new instruction is added to the slice.

Simple nodes are used to represent instructions not containing method calls, while *call site* nodes are used for instructions containing explicit or implicit[2] method calls.
Actual in and *actual out* nodes are used to represent values used or defined by a method at call sites. An actual in node represents a single value used by a method, such as a method parameter, an array, an object field or a static field. An actual out node represents a single value defined by a method.
Values used and defined by library methods are not represented using actual in and actual out nodes. A single *pseudo actual* node is used to represent all the values used and defined by a library method. In fact, since library methods are not analyzed in detail, there is no way to determine dependencies between output and input values. Using a single node to represent all of them provides a safe approximation, representing the fact that any output value could depend on any input value. Actual in nodes are only created when actual out nodes are examined.

[2]implicit method calls are calls to class static initializers

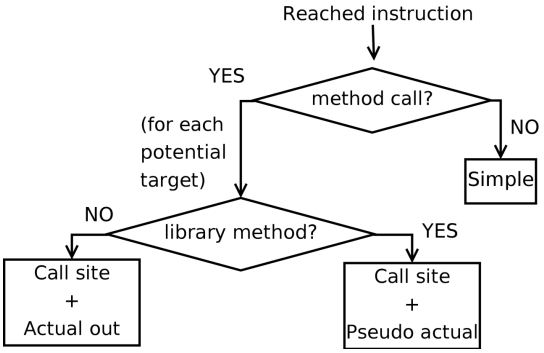

Figure 3: The flowchart for identifying pseudo-actual nodes

Nodes included in the slice are put in an *open list*. The algorithm extracts nodes from the open list one at a time, and executes different actions according to the node type (as shown in Table 1). For example, when a simple node is extracted from the open list, the algorithm examines data and control dependencies of the instruction represented by the node. Instructions on which the node depends are added to the slice, and corresponding nodes are added to the open list. However, when a call site node is extracted from the open list, control dependencies are examined as they are for simple nodes, but data dependencies are treated differently. In particular, the algorithm only examines data dependencies regarding the local variable on which the method is called. In fact, data dependencies regarding values used by the method are examined when the corresponding actual in nodes are examined. Once a node is examined, it is put in a *closed list*, which is used to avoid re-analyzing the same node.

2.3.3 Dependence relation

The most expensive part of the construction of the graph is the computation of summary edges (a detailed analysis of its cost is provided in [17]), that express the dependence of values defined by a method on values used by the same method. Graph-based slicing algorithms such as [7] require computing summary edges before the slicing phase begins. This can be very expensive in terms of required memory and computation time. For a typical Java program, the cost is $O(CallSites \times Params^3)$, where $Params$ is the maximum number of method parameters and $CallSites$ is the number of method call instructions in the code. Method parameters include object and static fields which are transitively used or modified by the method. Even for example programs using library methods, $Params$ is greater than 10,000 and $CallSites$ is greater than 100,000. This is why our algorithm computes and stores these dependencies during the slicing phase, using a dependence relation that is enriched as new dependencies are computed. When a method call instruction is put into the slice because of a data or control dependence, the computation of the dependence relation for

Node kind	Represents	Action
Simple	An instruction not containing a method call	Follow data and control dependencies
Call site	An instruction containing a method call	Follow control dependencies (and data dependencies for the local variable representing the receiving object)
Actual in	A value used by a method at a call site	Follow data dependencies for the value represented by the node
Actual out	A value defined by a method at a call site	Compute dependence relation, generate actual in nodes
Pseudo actual	All values used and defined by a library method at a call site	Follow data dependencies for every value used by the library method

Table 1: Actions corresponding to different kinds of slicing nodes

the related value is started. The value can be any value defined by the method, including thrown exceptions. To compute the dependence relation for a given value, the algorithm looks for the instructions that define the value. Then the algorithm starts examining dependencies according to table 1. During computation of the dependence relation, dependencies are used to reach other instructions. Values used by reached instructions are used to enrich the dependence relation. In fact, since the algorithm follows data and control dependencies, the values used by reached instructions influence the defined value. When the dependence relation is enriched, actual out nodes related to the examined method are analyzed again to create appropriate actual in nodes.

2.3.4 Current limitations

XCutter has currently some limitations. Some of them have effects on the correctness of the slice.

The Java language allows the programmer to call methods written in native languages, such as C and C++, using the Java Native Interface [18]. The slicing engine cannot analyze these methods, because there is no bytecode corresponding to them and thus Soot cannot create Jimple representations for them. Unfortunately, native methods might have side effects and not taking into account these side effects leads to incorrect slices. In the future we plan to add support for side effect specification of native methods.

Our slicer works under a closed world assumption. Some Java features do not respect this assumption, so our tool can not handle dynamic class loading and reflection, since they introduce in the program some elements unknown at compile time.

The slicing engine uses data and control dependencies to compute the slice. These two kinds of dependencies correctly describe sequential programs. To correctly take into account concurrent programs, however, other kinds of dependencies are needed. *Divergence dependencies*, *Interference dependencies*, *Synchronization dependencies*, and *Ready dependencies*, are used to model dependencies caused by synchronization and concurrency mechanisms [19]. The slicing engine does not consider these other kinds of dependencies, potentially and incorrectly excluding some instructions from the slice. However, data and control dependencies between instructions executed in the same thread are correctly taken into account.

2.3.5 Source code mapping

The computed slice is made of bytecode instructions, but it can be mapped back onto the source code, using source line information introduced by the weaver. Some instructions, however, are not correctly mapped. For example, most pointcut definitions are not mapped, because they generate no executable bytecode. In fact they are used by the weaver to identify join points where advice code has to be inserted. Another mapping problem is caused by *declare parents* or *introduce* instructions. These instructions are used by the weaver to modify the class hierarchy or the interface of the object oriented part of the program, but the weaver does not leave any trace of the modification in the bytecode, so these instructions are never included in the bytecode-level slice. To ease the work of bytecode analysis, we suggest that the weaver should put more information in the woven bytecode, exploiting, for example, the opportunity of annotating byte code introduced in Java5.

3. INTERFERENCE ANALYSIS

We exploit our slicer to study aspect interference. Consider the example shown in Listing 1.

The aspect `SpeedController` is interested in the calls to the "setters" of the `Factory` class: it regulates the speed, keeping it under a fixed value. The aspect `RotationMonitor` is in charge to log any speed change. While `SpeedController` modifies a property of the underlying system, the `RotationMonitor` is simply an observer. Thus, the `RotationMonitor` aspect does not interfere with the `SpeedController` one. (Conversely, the `SpeedController` does interfere with `RotationMonitor`).

We expected to be able to check this property with our slicer: a backward slice associated to `SpeedController` should not contain any of the statements of `RotationMonitor`. Unfortunately, things are more complicated. In fact, the dynamic nature of join-points selection means that the weaver has to put some machinery in the code. Modern weaver implementations use to translate each piece of advice as a method and to insert the translated code into the right point in the program, selecting a *Join point shadow* (i.e. the representation of a join point in the source code)[20]. They try, anyway, not to inline code to let the translated bytecode have the

Listing 1: The source code of two aspects to show interference definition

```
1  package examples.lollypop;
2
3  public aspect SpeedController {
4
5          pointcut speedset(Factory f,int x):
6                  call (public * Factory.set* (int) ) && args(x)
7                          && target(f);
8
9          after(Factory fact, int speed): speedset(fact,speed) {
10                 if (speed>4) {
11                         fact.setRotationSpeed(speed/2);
12                         System.out.println("check done");
13                 }
14         }
15 }
16
17
18 public aspect RotationMonitor {
19
20         pointcut speedmonitoring(Factory f,int speed):
21                 call (public void Factory.set* (int)) &&
22                 target(f) && args(speed);
23
24         after(Factory fact,int rpm): speedmonitoring(fact,rpm) {
25                 System.out.println("Lollypop stick rotation speed set to " + rpm + " rpms");
26         }
27 }
```

same accessibility rules than regular Java bytecode.
Following this approach it is sometimes necessary duplicating pieces of advice to translate properly a pointcut. An example of this behavior can be the *After Finally Advice*. This represents advice that should run after exiting from the selected join point, both in case of normal execution or in case of exception throwing. The translation strategy of the AspectJ compiler, in this case, is duplicating the call to the method that translates the given piece of advice. This implies the existence of a control dependence from the join point shadow to the advice methods call present in the normal execution branch and in the exceptional execution branch.
Moreover aspects are usually implemented following the singleton pattern (i.e. there is only one instance of each aspect in the system). The access to the aspect instance happens using the `aspectOf` static method of the aspect, that returns the required instance. This introduces a data dependence that is not present in the source code of the system.
An example of after finally advice translation is shown in listing 2. This listing shows the Jimple translation of the piece of advice of `SpeedController`. The statements at lines 26 and 42 are introduced by the translation of the after finally advice. To force the system to execute the piece of advice both in case of normal execution or in case of exception, at the end of the join-point shadow, is thrown an exception that is caught by instructions. in lines 57 and 58 introducing control dependencies that are not present in the source code of the system.
Lines 28 and 34 invokes the `aspectOf` method of `RotationMonitor`. This method returns an instance of the aspect itself. This is necessary since the piece of advice of `RotationMonitor` needs to execute after a speed change. The method `aspectOf` might throw a `NoAspectBoundException`. This exception can be caught at line 42, generating a control dependence from line 38 in listing 3 to the catch instruction at line 42 of listing 2.

These control dependencies, caused by the exception handling code introduced by the weaver, cause the interference analysis to assume that the two aspects interfere, even if, theoretically, we would expect no interference. Finding these dependencies is an important improvement in the accuracy of our prototype: our first version (described in [4]) could be successfully used to exclude interference between aspects like `RotationMonitor` and `SpeedController`, since it performed simpler, though potentially incorrect, analyses.
The spurious dependencies disappear if the pieces of advice shown in listing 1, which are of type `after finally`, are transformed into `after return` pieces of advice. In this case, the bytecode of the `SpeedController` aspect is simplified and does not use exceptions to manage control flow, as shown in listing 4.

There is no definitive solution to this problem since the dependencies are due to the semantics of the after finally advice. It should be, anyway, possible to ignore the dependency introduced by the translation of this kind of advice annotating, during the translation phase, the exceptions introduced. During the slicing phase the dependencies due to annotated exceptions can be ignored. This solution leaves unaltered the translated bytecode and does not alter the analysis semantics, since those exceptions, whose dependencies are ignored, are used only to transfer control.

4. LESSON LEARNED

Aspect oriented programming as popularized by AspectJ claims that cross-cutting concerns should be coded in iso-

Listing 2: The Jimple translation of the after advice of the aspect SpeedController in Listings 1

```
1  public class examples.lollypop.SpeedController extends java.lang.Object
2  {
3      public void ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory, int)
4      {
5          examples.lollypop.SpeedController r0, $r9, $r10;
6          examples.lollypop.Factory r1, r2;
7          int i0, i1;
8          java.lang.Throwable r3, r4, $r5, $r8;
9          examples.lollypop.RotationMonitor $r6, $r7;
10
11         r0 := @this: examples.lollypop.SpeedController;
12         r1 := @parameter0: examples.lollypop.Factory;
13         i0 := @parameter1: int;
14         if i0 <= 4 goto label7;
15
16         i1 = i0 / 2;
17         r2 = r1;
18
19     label0:
20         virtualinvoke r2.<examples.lollypop.Factory: void setRotationSpeed(int)>(i1);
21
22     label1:
23         goto label3;
24
25     label2:
26         $r5 := @caughtexception;
27         r3 = $r5;
28         $r6 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
29         virtualinvoke $r6.<examples.lollypop.RotationMonitor: void
30 ajc$after$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
31         throw r3;
32
33     label3:
34         $r7 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
35         virtualinvoke $r7.<examples.lollypop.RotationMonitor: void
36 ajc$after$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
37
38     label4:
39         goto label6;
40
41     label5:
42         $r8 := @caughtexception;
43         r4 = $r8;
44         $r9 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
45         virtualinvoke $r9.<examples.lollypop.SpeedController: void
46 ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
47         throw r4;
48
49     label6:
50         $r10 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
51         virtualinvoke $r10.<examples.lollypop.SpeedController: void
52 ajc$after$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
53
54     label7:
55         return;
56
57         catch java.lang.Throwable from label0 to label1 with label2;
58         catch java.lang.Throwable from label0 to label4 with label5;}}
```

Listing 3: The Jimple partial translation of the RotationMonitor aspect

```
25     public static examples.lollypop.RotationMonitor aspectOf()
26     {
27         examples.lollypop.RotationMonitor $r0, $r3;
28         java.lang.Throwable $r1;
29         org.aspectj.lang.NoAspectBoundException $r2;
30
31         $r0 = <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor ajc$perSingletonInstance>;
32         if $r0 != null goto label0;
33
34         $r2 = new org.aspectj.lang.NoAspectBoundException;
35         $r1 = <examples.lollypop.RotationMonitor: java.lang.Throwable ajc$initFailureCause>;
36         specialinvoke $r2.<org.aspectj.lang.NoAspectBoundException: void
37                 <init>(java.lang.String,java.lang.Throwable)>("examples_lollypop_RotationMonitor", $r1);
38         throw $r2;
39
40     label0:
41         $r3 = <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor ajc$perSingletonInstance>;
42         return $r3;}}
```

Listing 4: The Jimple translation of the after return advice of the aspect SpeedController

```
1  public class examples.lollypop.SpeedController extends java.lang.Object
2  {
3      public void ajc$afterReturning$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory, int)
4      {
5          examples.lollypop.SpeedController r0, $r4;
6          examples.lollypop.Factory r1, r2;
7          int i0, i1;
8          examples.lollypop.RotationMonitor $r3;
9
10         r0 := @this: examples.lollypop.SpeedController;
11         r1 := @parameter0: examples.lollypop.Factory;
12         i0 := @parameter1: int;
13         if i0 <= 4 goto label0;
14
15         i1 = i0 / 2;
16         r2 = r1;
17         virtualinvoke r2.<examples.lollypop.Factory: void setRotationSpeed(int)>(i1);
18         $r3 = staticinvoke <examples.lollypop.RotationMonitor: examples.lollypop.RotationMonitor aspectOf()>();
19         virtualinvoke $r3.<examples.lollypop.RotationMonitor:
20           void ajc$afterReturning$examples_lollypop_RotationMonitor$1$839313f3(examples.lollypop.Factory,int)>(r2, i1);
21         $r4 = staticinvoke <examples.lollypop.SpeedController: examples.lollypop.SpeedController aspectOf()>();
22         virtualinvoke $r4.<examples.lollypop.SpeedController:
23           void ajc$afterReturning$examples_lollypop_SpeedController$1$fda05aef(examples.lollypop.Factory,int)>(r2, i1);
24
25      label0:
26         return;
27     }
28
29 }
```

lation and woven automatically together. However, understanding interaction among different aspects is hard and tool support is still very poor.

Our experimental work shows that static analysis of woven code has some potential for making explicit the problems that arise due the complexity of intertwined code. However, simplistic slicing is not sufficient to determine whether two aspects may interfere. In fact, the machinery introduced *for the sake of the weaving itself,* makes slices always overlapping. Thus, our sufficient condition to exclude interference came out to be naive, since it is likely to be always false. Some of the dependencies, caused by the way advice weaving is performed, could be avoided with a parallel source level analysis or a suitable use of dynamic techniques. Smart heuristics are needed, though, and they are likely to depend heavily even on the lowest level of weaver implementation details. A better approach would be the use of annotations by the weaver itself, in order to keep track of the aspect oriented abstraction layer at the bytecode level.

Moreover, slicing Java bytecode also showed us that severe precision problems exist when real world programs are concerned. A common issue is due for example to library methods: consider two calls to the `add` method of two different `Vectors`. Unless the slicer creates multiple copies of the same method to distinguish among different receiving objects, the static analysis will detect spurious dependencies, resulting in large slices. Native code is almost ubiquitous in library frameworks and this means that some dependencies may also be neglected: big slices can even be incomplete! Static analysis of bytecode should be used as a support to further analyses at different levels. Furthermore, the closed world assumption behind any static analysis is challenged by current coding practice. Dynamic linking and reflection are common place in most applications. However the expressive power of intertype declarations common in AspectJ programs forces any analysis to take into account every aspect unit just to compute the static structure of the type system.

The path towards having crosscutting components that can be safely plugged into a system is still long. AspectJ aspects make easy to program quick pools of sparse code and their use spread among developers. However, the next step in dealing with complex cross-cutting concerns and their interaction and evolution needs at least a better tool support.

Acknowledgments

The authors would like to thank Davide Balzarotti and Carlo Ghezzi for their help in clarifying the ideas behind this paper.

5. REFERENCES

[1] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241*, 1997.

[3] Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, 2004.

[4] Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. Slicing AspectJ Woven Code. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2005*, 2005.

[5] Mark Weiser. Program slicing. *In: IEEE Transactions on Software Engineering, vol. SE-10, no. 4. pp. 352-357*, July 1984.

[6] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, 1984.

[7] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.

[8] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[9] Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.

[10] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent Java programs using Indus and Kaveri, 2006. Tech report. Available at `http://projects.cis.ksu.edu/docman/view.php/12/117/stt05-submission.pdf`.

[11] Santos Academic License. `http://www.cis.ksu.edu/santos/KSUAcademicLicense.shtml`.

[12] Raja Vallée-Rai. Soot: a Java Bytecode Optimization Framework. Master's thesis, McGill University, July 2000.

[13] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[14] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 2002.

[15] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press.

[16] Neil Walkinshaw, Mark Roper, and Murray Wood. The Java System Dependence Graph, September 2003. In Third IEEE International Workshop on Source Code Analysis and Manipulation, page 55.

[17] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[18] Sheng Liang. *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[19] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium*, pages 1–18, 1999.

[20] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD'04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.