Using Program Slicing to Analyze Aspect Oriented Composition

Davide Balzarotti
Dip. Elettronical e Informatica
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
balzarot@elet.polimi.it

Mattia Monga
Dip. Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41
20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

AspectJ language was proposed to make cross-cutting concerns clearly identifiable with special linguistic constructs called aspects. In order to analyze the properties of an aspect one should consider the aspect itself and the part of the system it affects. This part is just a slice of the entire system and can be extracted by exploiting program slicing algorithms. However, the expressive power of AspectJ constructs forces slicers to take into account big portions of programs. We suggest that AspectJ should regulate more formally the interaction among code units, by defining some stricter boundaries around aspect influence, otherwise the separation turns out to be just syntactic sugar.

1. INTRODUCTION

Aspect oriented languages claim to be able to provide linguistic support to make cross-cutting concerns isolated in proper code units. Currently the most successful aspect-oriented language is probably AspectJ [17]. Designed and implemented at Xerox PARC, it is aimed at managing tangled concerns in Java programs.

AspectJ provides first-class entities called **aspect**s that, in a strong analogy to regular Java **class**es, can define fragments of code called *advices* that will be woven at run time **before**, **after**, or **around** interesting points (*join points*) in the whole program. AspectJ showed up to be very convenient to express cross-cutting concerns. A typical AspectJ advice can be something like "before any call to the division function, check if the divisor is not zero"; in a very economical way it is possible to affect all the divisions in the code, even without knowing where these divisions will occur.

The problem we want to discuss in this paper is if the aspect oriented computation is actually separated from the rest of the program by using the linguistic construct provided by AspectJ. In fact, on the one hand a "no division by zero" aspect would be a isolated code unit. However, on the other hand it might be difficult to figure out the behavior of the whole system: every time the division function is called, one has to consider that also the aspect oriented code is executed. In order to asses the resulting complexity of an aspect oriented program, we tried to apply well known techniques of program comprehension, namely static analysis and program slicing, to AspectJ.

In the rest of the paper we describe the results of our preliminary experiments. The discussion is organized as follows: in Section 2 we briefly introduce program slicing techniques, in Section 3 we propose our approach to slice aspect oriented programs, in Section 4 we examine the problems we found, and finally in Section 5 we draw some conclusions.

2. ASPECT ORIENTED PROGRAM SLICING

Program slicing was proposed by Weiser [16] in the early 80's. It is a technique aimed at extracting program elements related to a particular computation. A *slice* of a program is the set of statements which affect a given point in a executable program (*slicing criterion*). One can compute statically the set of statements that potentially affect the slicing criterion for every possible program execution (static slicing), or one can consider the information about a particular execution of the program and derive a dynamic slice [3] of a program.

Different slicing algorithms and different type of slice have been proposed by many authors [14, 5]. Nowadays a widely adopted approach to compute static program slicing consists in re-formulating the problem as a reachability problem on a particular graph representation of the program, which, for inter-procedural slicing, is the so called *system dependencies graph* (SDG) [7]. It is worth noting that, while producing the minimal slice is known to be uncomputable, it is possible to compute non-minimal slices with fairly efficient algorithms.

Program slicing was initially studied for procedural programming language but has then be extended to cope with the object oriented paradigm by Larsen and Harrold [9, 10] (specific solutions for Java language are proposed in [11, 8, 18]). Notwithstanding the rich theoretical work done in the

field, the only publicly available tool we are aware of which is able to compute a program slice of a Java program is Bandera [4].

The application of program slicing techniques to aspect oriented software is a novel research topic.

A preliminary work in this area has been done by Zhao [19]. He proposed an aspect-oriented SDG that is a further extension of the object-oriented SDG. The aspect-oriented system dependence graph (ASDG) consists of an SDG for the traditional code enriched with a set of dependence graphs that represent the aspect code. Graphs are connected through special edges that model introductions and advice execution. He focused on AspectJ, however he did not consider that aspect advices might apply to the aspect oriented code itself.

We start analyzing the problem of slicing aspect oriented in [2]. We initially proposed an approach based on the concept of *conjugated class* of an aspect to allow the application of existing object oriented algorithms. A conjugated class contains all members and methods of the originating aspect and it has a method for each advice. We did not propose a complete solution of the problem, since we did not describe how conjugated class should be connected to the rest of the program and we were not able to cope with introductions and other subtilties of AspectJ syntax. In the next Section, we describe the new approach we think is most suitable to implement a real tool able to slice a larger family of AspectJ programs.

3. SLICING JAVA BYTE-CODE

Since AspectJ programs are eventually woven in Java byte-code binaries, which are executed by a Java Virtual Machine, in order to slice them two different approaches are possible: (1) one can consider methods and advices as first-class entities and try to extend SDGs to take them into account [2, 19, 13]; otherwise, (2) one can try to analyze the woven program by applying existing techniques and map the results on the original structure of the program.

The high-level approach is conceptually more appealing, since it does not depend on the actual implementation of the AspectJ weaver, and, more fundamentally, it enables the use of aspects as first-class entities in the resulting model. However, building a working tool is far from trivial, because it needs to be able to manage a several AspectJ syntax details. In particular, the AspectJ pointcut definition language allows programmers to characterize pointcuts on a wide range of abstraction levels:

- Lexical (**withincode**, regular expression on identifiers, etc.)
- Statically known interfaces (**void** *.`func`(**int**), etc.)
- Run time events (**call**, **execution**, **set**, **if**, etc.)

In order to build as quick as possible a tool for experimenting with and slicing real world programs, we adopted a more pragmatic strategy:

1. Compile classes and aspects using the AspectJ compiler.
2. Weave aspects into an executable program.
3. Apply existing slicing algorithms (we built upon the Soot static analysis framework [15]) to the resulting byte-code.
4. Obtain a slice, as a set of byte-code statements.
5. Map the results onto the original aspect oriented source code.

Working at the level of Java byte-code could appear not appropriate because any distinction among classes and aspects may seem to be lost. The AspectJ weaver translates aspects in classes, advices in methods, and join points in methods invocation. Thanks to this approach, it is not difficult to map every statements to its original aspect (or class). However, a tool based on byte-code slicing has to be changed when the AspectJ weaver modifies its implementation strategy.

Figure 1 shows an example that contains only a trivial class `C` and an aspect `A`. The aspect introduces a public field into the class and it defines an advice that print a value when `method2`() is called. The resulting woven classes are shown on the right hand side. The aspect has been translated in a class and the advice in an equivalent method. A decompilation of `C.method`() produces:

```
...
 3 invokevirtual #17 <Method void method2()>
 6 invokestatic #31 <Method A aspectOf()>
 9 aload_1
10 invokevirtual #34
      <Method void ajc$afterReturning$A$21(C)>
...
```

It is easy to identify the call to after-returning advice after the invocation of `method2`().

Thus, by using dedicated libraries it is fairly easy to build a tool for inspecting the Java byte-code, obtaining the call graph, and performing the def-use analysis. It is then possible to implement existing algorithms to construct the system dependence graph and to calculate static or dynamic slices.

4. ANALYSIS OF ASPECT INTERACTION

The final goal of our work is to be able to analysis interactions among aspects. An aspect oriented program is composed by weaving aspect and classes together. An aspect is conceptually *a posteriori* with respect to the rest of a system. When a programmer writes a new aspect, s/he assumes that the rest of the system is working correctly and s/he hopes to add the new cross-cutting functionality *without breaking the system*. How one can check that the new aspect does not interfere with existing aspects and classes?

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

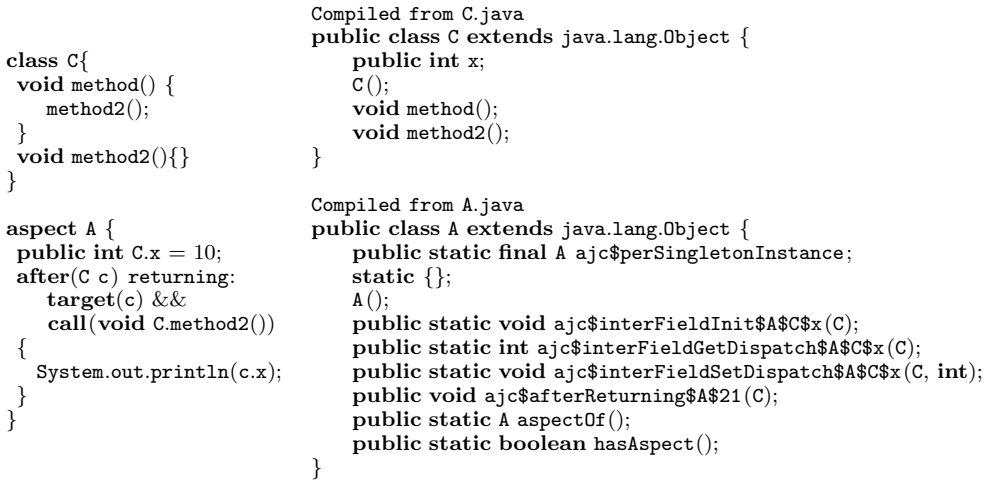

Figure 1: A simple class before and after the weaving of an aspect

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 $x \leq 0$.

In [2] we proposed the following sufficient condition to check non-interference between aspects:

> Let A_1 and A_2 be two aspects and S_1 and S_2 the corresponding backward and forward slices obtained by using the pointcuts declarations defined in A_1 and A_2 as slicing criteria. A_1 does not interfere with A_2 if
>
> $$S_1 \cap S_2 = \emptyset$$

This condition may be too strong: in fact, two aspects may not interfere also if their slices share some statements.

A weaker and more practical condition is

> Let A_1 and A_2 be two aspects and S_1 and S_2 the corresponding backward slices obtained by using all the statements defined in A_1 and A_2 as slicing criteria. A_1 does not interfere with A_2 if
>
> $$A_1 \cap S_2 = \emptyset$$

S_2 contains all statements that affect the slicing criterion (which contain all the statements of A_2). It is worth noting that the interference relation is not symmetric. In fact, it is possible that A_1 interferes with A_2 but A_2 does not interfere with A_1. For example, if A_2 is a tracing aspect and A_1 is an aspect that change the order in which procedures are called, the application of A_2 does not change an existing A_1, but the application of A_1 onto A_2 change its behavior.

Moreover, modifications in the type hierarchy can be difficult to deal with. Consider the following program:

```
class ClassA{
  void method() {}
  public static void main(String[] args) {
   ClassA a = new ClassA();
   a.method();
  }
}

class ClassB {}

aspect A1 {
  declare parents: ClassA extends ClassB;
}

aspect A2 {
  before(): call(* ClassB+.*()) {
   // ....
  }
}
```

If we apply the aspect `A2` but not the aspect `A1` the advice is never executed. When we add to the program the aspect `A1` the advice is executed before the invocation of `ClassA.method()`. By using **declare parents**: we forced the members of `ClassB` to become part of `ClassA` also.

4.1 On the Precision of Slicing

A well-known result asserts that, in general, the problem of finding the minimum static slice is incomputable [16]. This means that when we compute a slice, the result may contains some unnecessary nodes. This is not a problem in most cases but sometime can lead to an incorrect result.

For interference analysis, Figure 2(a) shows how a non-minimum slice might cause a decision error when we analyze the interference between aspects. The result is always correct if the algorithm does not find any intersection between the two sets, but could be wrong if an intersection occurs.

Another problem that could have a significant effect on slice precision is the resolution of aliases. An alias occurs when two or more different variables refer to the same memory location. The computation of slices relays on the construction of the SDG, that in turn requires to calculate control

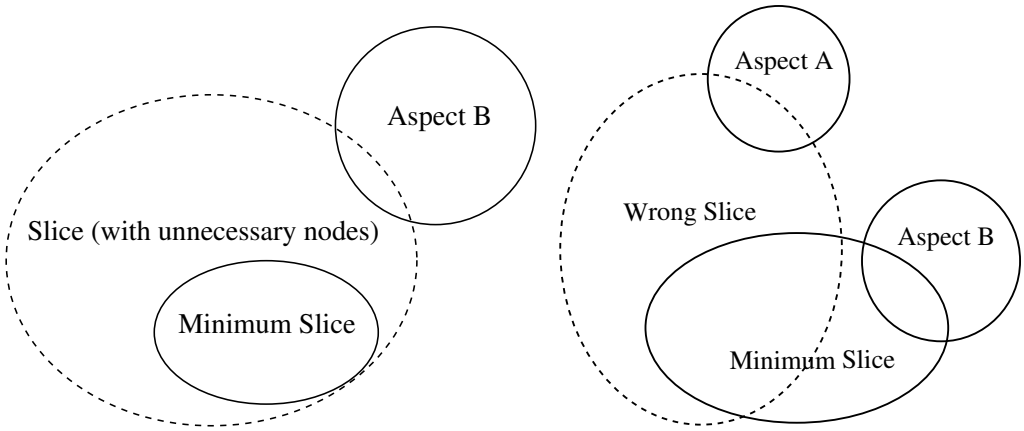

Figure 2: Incorrect results in presence of non-correct slice

and data dependencies among every program statement. In presence of aliasing the exact computation of data dependency becomes a very difficult task [12]. In order to mitigate the impact of aliases it is necessary to adopt a generalization of the notion of data dependence [1]. If the algorithm does not correctly take into consideration all may-alias variables (note that this is possible only assuming a closed world hypothesis), the resulting slice may omit some required statement. Thus, in general we might find a slice that contains statements that are unnecessary (since we cannot compute the minimum slice) and omits others (since pragmatic issues could force us to use approximate algorithms). Figure 2(b) shows an example in which the slice identifies an incorrect interference with the aspect A and does not identify the correct interference with the aspect B. Therefore, in order to keep significant our non-interference criterion, we have to adopt a conservative approach that guarantees that every possible may-alias are take in consideration. This resolve the problem of false negative (when the algorithm does not find an existing interference) but it tends to make the slice bigger, increasing the number of false positive (when the algorithm finds a false interference).

Moreover, some AspectJ join points are not statically determinable. AspectJ provides some primitives to declare pointcuts that discriminates based on the dynamic context (i.e. **cflow**, **cflowbelow**, **if**, **this**, ...). A conservative approach requires to consider every possible execution trace, but this may further increase the number of false-positive response.

Adding together all the previous considerations, the final precision of a hypothetic tool that analyze aspect interaction using static slicing techniques may become quite low. In general, program slicing can be used to automatically build an abstraction (i.e., a simplified model) of a program. Often the goal is to reduce the dimension of a program in order to reduce the complexity of algorithms that are exponential in the number of program statements, for example in order to be able to apply a model checking tool [4]. In this case, the size of slice is a minor issue, because every discarded statement is a benefit anyway. However, when one is interested in deciding if a given statement belongs to a slice, the dimension of the slice becomes a major issue, because the accuracy of the decision depends on it.

5. CONCLUSIONS AND FUTURE WORK

In object oriented programs one can define *composition invariants* which are properties of classes that are preserved in every possible composition of objects. Software engineers leverage on this in order to reason on the properties of whole object oriented systems without considering all the details.

Unfortunately, in general no composition invariants are guaranteed to hold anymore, when AspectJ is in use. We tried to derive the slice of a system affected by an aspect, but the loosely regulated expressiveness of AspectJ constructs causes a turbulent *ripple effect* that in general forces to take into account most of the statements of a system. Moreover, a whole system analysis is needed, because arbitrary introductions and modifications of type hierarchy require a closed world assumption to be resolvable. This is in part due to the *obliviousness* that Filman identifies as an intrinsic characteristic of aspect orientation [6], but it is amplified by the undisciplined power of AspectJ constructs. The ultimate goal of aspect-oriented programming is the separation of otherwise cross-cutting concerns. However, these benefits are lost if the comprehension of aspect properties entails the analysis of the whole program. Instead, if we would be able to define some boundaries around aspect influence, the separation turns out to be not just syntactic sugar but a true aid in dealing with program complexity.

Currently we are improving our tool for slicing Java bytecode that we want to apply to AspectJ code. We are interested in evaluating the actual impact of AspectJ constructs in real world aspect oriented system. Our final goal is to define patterns of use and/or new AspectJ constructs to improve the comprehensibility and maintainability of AspectJ programs.

6. REFERENCES

[1] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances of Computing*, 43:1–50, 1996.

[2] L. Blair and M. Monga. Reasoning on AspectJ programmes. In *Proceedings of Workshop on Aspect-Oriented Software Development*, pages 45–50, Essen, Germany, Mar. 2003. German Informatics Society.

[3] J. W. L. Bogdan Korel. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[5] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[8] G. Kovcs, F. Magyar, and T. Gyimthy. Static slicing of Java programs.

[9] L. Larsen and M. Harrold. Slicing object-oriented software. In *In Proceedings of the 18th International Conference on Software Engineering*, pages 45–50. Association for Computer Machinery, Mar. 1996.

[10] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[11] M. W. Neil Walkinshaw, Marc Roper. The Java system dependence graph. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, page 55, Sept. 2003.

[12] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. Technical Report GIT-CC-00-33, College of Computing, Georgia Institute of Technology, Dec. 2000.

[13] M. Stoerzer. Analysis of AspectJ programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, Mar. 2003.

[14] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[15] R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a Java bytecode optimization framework. 1999.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, Mar. 1981.

[17] XEROX Palo Alto Research Center. *AspectJ: User's Guide and Primer*, 1999.

[18] J. Zhao. Applying program dependence analysis to Java software. In *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, December 1998.

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