Slicing AspectJ Woven Code

Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5, I-20133 Milano, Italy

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39/41, I-20135 Milano, Italy,

ABSTRACT

The AspectJ programming 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 makes difficult to implement slicing algorithms that are both precise and produce useful, i.e. small enough, slices. In this paper we describe our approach to slice AspectJ programs, based on the analysis of the woven code.

1. INTRODUCTION

Well organized software systems are partitioned in modular units each addressing a well defined concern. Such parts are developed in relative isolation and then assembled to produce the whole system. A clean and explicit separation of concerns reduces the complexity of the description of the individual problems, thereby increasing the comprehensibility of the complete system [15].

The notion of aspect-oriented programming was introduced by Kiczales et al. in [10]. Their approach was successfully implemented in AspectJ [1] by Xerox PARC. AspectJ aims at managing tangled concerns at the level of Java code. AspectJ allows for definition of first-class entities called `aspects`. These constructs are reminiscent of the Java `class`: it is a code unit with a name and its own data members and methods. In addition, aspects may *introduce* an attribute or a method in existing classes and *advise* that some code is to be executed `before` or `after` a specific event occurs during the execution of the whole program. Aspect definitions are *woven* into the 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FOAL 2005 Chicago, Michigan USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

The AspectJ way to provide support for encapsulating otherwise cross-cutting concerns is based on:

1. a syntactic extension to Java for isolating aspect-oriented code;
2. a language for identifying *join points* where advice code should be introduced; set of join points are called *pointcuts*
3. a compile time *weaver* responsible to mix aspects with the rest of the code in order to produce the running system.

The constructs provided by AspectJ show 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. However, it is not clear how real the separation is. In fact, even though a "no division by zero" aspect would be a isolated code unit, 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 general, aspects, while coded in a separate unit, do not enable a true modular reasoning[5, 16]. Moreover, it is still not clear how to cope with the difficult problem of aspect interaction (see [7, 9, 3, 12] for some work in progress and discussion). 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 this paper we describe our effort for building a slicer able to identify which part of an AspectJ program is affected by a specific aspect.

The paper is organized as follows: in Section 2 we present the challenge of slicing AspectJ programs, in Section 3 we describe our approach to the problem, in Section 4 we sketch the implementation of our tool, in Section 5 we show a simple example, and finally in Section 6 we draw some conclusions.

2. SLICING AO PROGRAMS

Program slicing is a program analysis technique introduced by Weiser in the first half of the '80s [21]. A backward (or forward) *slice* of a program consists in all the statements that may influence (or may be influenced by) a given set of statements, called the *slicing criterion*.

We will focus our attention on backward slicing based only

on static information, i.e., without making any hypothesis about input data. Slicing techniques were initially proposed for procedural programs, however they have been widely studied and applied also to object oriented programs [13]. In [14] Liang and Harrold approached the slicing of object oriented programs as a graph reachability problem: each method in an object oriented program is represented by a directed graph (*method dependence graph*, MDG) in which every statement is a node and edges represent control and data dependences among them. All MDGs are then merged in a system dependence graph (SDG), a directed graph that represents the whole analyzed program. On this graph, slices can be computed by exploiting the algorithm introduced by Horwitz, Binkley, and Reps [8].

Unfortunately, the techniques developed for object oriented languages cannot be used as they are with aspect oriented languages, due to some specific aspect oriented features present in most modern aspect oriented languages. In fact, AspectJ provides powerful constructs that, while giving great power to programmers, pose a number of problems during static analyses of the code.

A first issue is the use of "inter-type declarations". In fact, aspects can modify a type by introducing a member in a class or even by manipulating the type hierarchy. This might be useful when one wants to adapt an existing class to a given interface. The use of these constructs, though handy in most cases, has the evil effect to force any analysis to be "holistic", because every analysis needs a closed world assumption. If one wants, for instance, to decide if between two classes `A` and `B` an inheritance relationship holds (a critical information needed when examining polymorphic calls), one has to analyze all the aspects, because any of them could declare such a relationship. Similarly, point-cuts may be defined by using wildcards. This flexibility forces the analysis to take into account all the code of the system.

Another issue is the use of dynamic properties in pointcut definitions. For example, Figure 1 shows a pointcut definition that depends on the value of the function `If.getInputFromUser()`.

```
aspect Trace{
    before(): call(void System.exit(int))
        && if(If.getInputFromUser()){
         System.out.println(thisJoinPoint);
    }
}
```

Figure 1: A pointcut definition that uses dynamic properties

Thus, specific slicing techniques for AspectJ programs were proposed. Zhao and Rinard proposed an algorithm for building a system dependence graph specific for AspectJ programs [22]. They consider each advice like a method and associate an MDG to each of them. Two cases are considered for inter-type declarations. If an inter-type declaration introduces a method it is represented using a module dependence graph. If it introduces a field in a class it is considered as an instance variable of both the aspect, that introduced the field, and the class in which it is introduced.
A pointcut is represented with a *join point vertex.* A *weaving arc* connects the point in the Java part of the AspectJ program picked up by the pointcut to the join point vertex. The join point vertex is connected to the module dependence graph entry vertex associated with it.
Eventually, the whole aspect is represented by an *aspect dependence graph*, a directed graph whose entry vertex is connected by an *aspect membership arc* to the join point vertices and the module dependence graphs declared by the represented aspect. The aspect dependence graph represents the parameters, eventually passed or used by advices or inter-type declared methods, with formal in and formal out vertices, just like formal vertices used in Liang and Harrold's system dependence graph.

3. AO SLICING OF WOVEN CODE

Slicing AspectJ programs by considering methods and advice code as first-class entities [4, 22, 18] is conceptually 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 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.)

For example, [22] does not take into account wildcards, changes in class hierarchy, and dynamic pointcuts. Also whether it would be possible to manage all these characteristics with ad-hoc (and not easy to implement) solutions, the resulting program should implement a lot of features currently implemented by the AspectJ compiler.

Instead, one can try to analyze the woven program, i.e., plain Java bytecode, by applying existing techniques and map the results on the original structure of the program.

Thus, in order to build as quick as possible a tool for experimenting with AspectJ 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 [19]) 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.

The advantage in adopting such an approach is twofold. First, it is not necessary to translate aspects into classes because this task is done (in a better way) by AspectJ itself. Second, this approach does not neglect any detail related to AspectJ syntax and it does not need any modification in case of changes in some AspectJ functionalities.

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 statement 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.

Thus, the strategy we implemented in our tool starts by inspecting the Java byte-code, then the call graph is computed and the "def-use" analysis performed. Eventually, an SDG of the woven Java program is built, and, by exploiting standard algorithms proposed by the program analysis community during the last 20 years, static slices are computed. Finally, slices can be mapped backwards to the AspectJ code, leveraging on the information about aspects that is still encoded in the byte code. In the following section we describe how the tool was implemented and the limitations of the current prototype.

4. SLICER IMPLEMENTATION

4.1 Strategy and limitations

Notwithstanding the deep research work done in the slicing field, only a few products able to do slicing of real object-oriented programs exist: for example, the Bandera tool [6] has a component aimed at slicing Java programs in order to ease model checking of properties of multi-threading programs. Bandera operates at the bytecode level, thus one could imagine its use also in an AspectJ context (remember that AspectJ programs are eventually woven in plain bytecode). However, all our attempts to use it for slicing programs generated by the AspectJ compiler failed, since the code, while publicly available under a GPL license, is hard to understand and evolve. In fact, Bandera's slicing component is targeted to slice synchronization constructs, thus, it should be enhanced to deal with generic slices. Recently, Bandera's research group released the Indus program slicer [2]. We did not test the new tool, yet. However, using the Indus tool required a licence agreement conflicting with our goal of producing an open source tool.

However, both Bandera and Indus are based on a Java program analysis framework called Soot [19][1]. Therefore, we decided to build our slicer directly on the Soot set of libraries. The Soot framework uses an intermediate representations called *Jimple*, that simplifies the analysis of the bytecode. The algorithm used to compute slices is the one proposed by Horwitz, Binkley and Reps [8]. This algorithm works on the SDG of the program that is built by putting together the MDGs of all methods. It is worth noting that the advice code is represented by plain methods in the woven code, thus normal techniques apply. In order to build the SDG, each method is analyzed after the methods it calls. For each analyzed method the following graphs are computed

- the control flow graph (CFG), representing the control flow among statements: a statement a is connected to b if b can be executed immediately after a;
- the control dependence graph (CDG), representing statement dependences from conditional statements;
- the data flow graph (DFG), representing data dependences: a statement a is connected to b if b uses a variable defined by a.

[1]Soot itself is LGPL, thus there is no contradiction in using it in a proprietary tool as Indus is.

The main purpose of our work was to show the feasibility of our slicing approach based on bytecode analysis. We would like to have a tool as quick as possible to start experimenting with AspectJ examples of increasing complexity. In order to build a working tool in a reasonable amount of time, the current prototype has the following limitations:

- variables are not of array types;
- no exception handling mechanism is used;
- there are no inner classes;
- there are no static members;
- there are no recursive calls;
- the program has a single thread of control;
- there are no inter-procedural aliases.

The tool produces correct slices (i.e, slices that contain all the statements that might be part of the minimal slice, that is not computable [20]) for any program that satisfies the above limitations. We intend to remove all these limitations in the forthcoming versions of the tool, but their introduction was useful to build quickly a proof-of-concept prototype. Our tool is available on request under a GPL license agreement.

4.2 Building the graphs

In order to build the CFG and the CDG of each method we relied on the Soot framework. To each CFG we added a *Start* node to represent the method's entry point and a *Stop* node to represent the method's exit point. Then the CDG was built following the approach used by the Bandera tool [6].

Since a method (or advice code) often calls another method (or advice code) their MDGs must be connected at the call sites. An example showing how call sites are connected is shown in Figure 2.

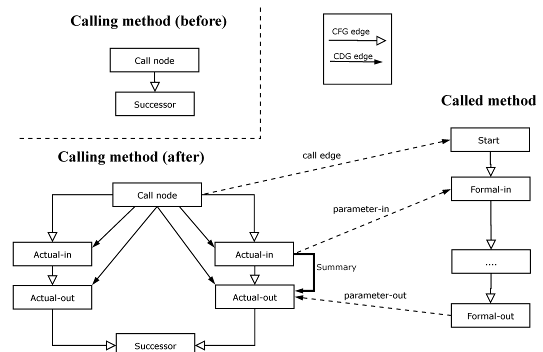

Figure 2: Call site modification for a polymorphic call

Due to polymorphism, the piece of code actually executed could be decided only at run-time, thus we created a new branch of control flow for each possible call target. In each branch, we put actual-in and actual-out nodes depending on

how parameters are used by the called method.
We say that a method *uses* one of its parameters if it reads the parameter value or if it defines its value. We say that a method *defines* one of its parameters if it defines the parameter value. We put an actual-in node for each parameter used by the callee and an actual-out node for each parameter that may be modified by the callee.

We chose to analyze each method after the methods it calls, so we know which parameters are used or defined and we can determine which actual nodes we have to create.
To take into account dependencies among actual and formal parameters, we added some edges to the SDG. We put a parameter-in edge from an actual-in node to the corresponding formal-in node in the called method. We put a parameter-out edge from a formal-out node in the called method to the corresponding actual-out node in the calling method.

Actual-in and actual-out nodes in each branch are control dependent on the method call instruction, so we add control dependence edges from the call node to actual-in and actual-out nodes. To take into account the dependence of called method on its caller, the call node itself is linked to the entry vertex of called method with a method call edge. Since method call edges are interprocedural edges, they are only put in the SDG.
Next we add summary edges from actual-in nodes to actual-out nodes. A summary edge is added from actual-in A to actual-out B if and only if the value of A may affect the value of B. Again, these dependencies have been computed during the analysis of the called method.
The last node of each branch is eventually connected with a control flow edge to the original call node successor in the CFG. Figure 2 shows a modified call in case there is a single parameter and its value is modified by the called method.

4.2.1 Formal Parameters and Return Value Representation

After the creation of actual nodes, we have to build formal-in and formal-out nodes to represent formal parameters of the method under analysis. Since Jimple representation already contains instructions representing the assignment of parameter values to local variables, we use these instructions as formal-in nodes.

To create formal-out nodes, we analyze method instructions one by one, searching for instructions that modify reference-type parameters. Non-reference parameters (primitive types as **int**s) cannot be modified, since their redefinition is not returned to the calling method, so we do not create formal-out nodes for them. For each parameter, we add a formal-out node in the method if and only if there is at least one instruction that can modify the parameter. Formal-out nodes are placed sequentially before the Stop node in the CFG of the called method. To handle multiple return statements, we link each one of them to the first formal-out node (if there are no formal-out nodes, they are linked to the Stop node).

4.3 Dataflow analysis

DFGs are built following a slightly modified version of the algorithm proposed in [17]. This algorithm requires to associate six sets (*use*, *def*, *gen*, *kill*, *in*, and *out*) to each node. The *def* set contains the variables defined in a given node. In our implementation the *gen* set (generated from the *def* one) contains strings in the form 'node-number.variable-defined'. For example, if node 7 defines variable 'foo', we put in node 7 *gen* set the string '7.foo'. When we find a killing definition[2] of 'foo', we put in the killing node *kill* set the string '*.foo'. This means that every other definition of 'foo' has to be killed. We also use the character '*' to express datamember killing. When we find a killing definition of the reference variable 'bar', we put in the killing node kill set the string '*.bar.*', instead of explicitly killing all its data-members. This allows us to represent killing definitions for object data-members without knowing the inner structure of classes.

Computation of reaching definitions needs comprehension of intra-procedural alias information. We don't describe here the algorithms we use to compute intraprocedural aliases, since they are performed by the Soot framework. Since aliases affect variables uses and definitions in method nodes, we have to modify *gen*, *def* and *use* sets for each node.

For each used variable in the node we build a graph to express the use of its 'containers' and its aliases. The 'container' of a class datamember is the object the datamember belongs to. An example of this graph is shown in figure 3.

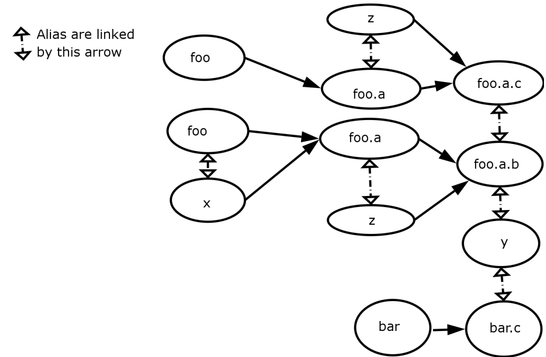

Figure 3: ***Alias graph example***

We start building this graph by adding the used variable to it. Next we add to the graph its aliases. Then, for each graph head, we add to the graph its 'container', linking the container with the head. When we add a 'container' to the graph, we also add its aliases and we link them with the head we are examining. We go on until no more containers can be added to the graph.

From this graph we can extract the node *use* set. We examine the graph heads one by one. We follow graph edges until we reach one of the tails. For each node we come across, we take the last part of its name and concatenate it with others. The obtained variables are added to the node use set. Then we remove the examined head from the graph.
Referring to figure 3, we can examine what happens when examining the head named 'x'. First, 'x' is added to the node use set. Then we follow the edge and we come across 'foo.a'. We add 'x.a' to the node use set. Next we come across 'foo.a.b'', and we add 'x.a.b' to the node use set. Since we have reached a tail node, we remove the 'x' node from the graph.
If we are examining a variable being defined by the node, we build the graph in the same way, except that we do not add

[2]Killing definitions are definitions that make useless previous definitions of the same variable

variables alias. However containers' aliases are still added to the graph.

Reaching definitions are then computed using the same algorithm proposed in [17], with some minor modifications needed to make it work with our representation of killing definitions.

The last step in the method analysis consists in calculating dependencies of formal-out nodes and return value node from formal-in nodes. Starting from each formal-out node, we backwards follow summary, control and data dependence edges, looking for formal-in nodes. Any formal-in node found in this intraprocedural slice affects the formal-out node we examined.

4.4 Mapping the Slice to the AspectJ Code

Once the slice of code that affects a given criterion is computed, we have to map it back onto the original AspectJ source code. This is accomplished analyzing source code information found into the bytecode instructions. Since bytecode contains information about original source code lines corresponding to each bytecode instruction, the mapping is performed extracting source code line numbers from bytecode. In this way method and advice statements are easily identified. Instead, pointcut declarations do not normally contain executable statements. However, when they do (as in the example shown in Fig. 1), the mapping is solved correctly.

Currently, we are not able to correctly map inter-type declarations. In fact, AspectJ weaver documentation does not precisely describe the weaving of inter-type declaration (that part is prone to heavy optimization, and it is likely to change in different releases). Inter-type declarations are implemented by direct bytecode manipulation, without preserving any information about their source, therefore by analyzing only bytecode it is impossible to spot them correctly.

5. AN EXAMPLE

Figure 4 contains a piece of code that shows a simple case of aspect interference. Class `T` represents an hypothetical boiler controller. Suppose the programmer wants to add two different aspects. The first one (`LockAspect` in the code) introduces a locking mechanism in order to assure that only one object at a time can modify the boiler status. The second aspect (`TInvariant` in the code) checks that the boiler temperature can never be set to a value greater than 100 Celsius degrees and shuts down the boiler otherwise. Both the aspects work properly if they are independently applied to the program but if they are applied together the invariant aspect can in some case interfere with the locking mechanism leading the system to a deadlock status.

Applying our tool to the weaved bytecode it is possible to construct the SDG and then calculate the slices using the two aspects as slicing criteria. The whole graph contains 236 nodes and around 650 edges.

The slice built using `TInvariant` as slicing criterion does not contain any node that belongs to the `LockAspect` code. That means that the locking mechanism does not interfere with the invariant property. On the contrary, the slide built starting from `LockAspect` contains nodes of the `invariant` code. This is not a proof that the two aspects are incompatible, but it represents a useful information for the programmer since it points out that the `TInvariant` aspect affects the behavior of the locking aspect.

6. CONCLUSIONS

Our tool, while quite limited in the current preliminary version, shows that AspectJ programs analysis can be actually performed analyzing woven bytecode. Moreover, since every Java program is also a valid AspectJ program, the tool can also be used to analyze plain Java programs, provided that the limitations described in Section 4.1 are satisfied.

Mapping the computed slice onto the source code is currently possible only for statements that are part of methods and advice code. The main open problem is still about the inter-type declarations: currently they are not correctly mapped, because it is not easy to understand whether class files have been modified during the weaving process. However, we can correctly analyze effects of inter-type declarations in the program. Mapping of inter-type declarations would be easily implemented if AspectJ compiler could mark intertype declarations using Java annotations, provided by Java 1.5.

We plan to remove most of the limitations of the tool in the following releases. We will be able soon to analyze arrays introducing in our code the opportune representation. Soot already provides a representation for arrays and we are going to use it. We will also implement a simplyfied exception analysis, that will be able to deal with intraprocedural exceptions, using tools provided by Soot.

The further step will be dealing with direct and mutual recursion along the lines sketched in [17]. Morever, static fields will be analyzed by exploiting the techniques described in [11].

Our final goal is to understand how large is the impact of using aspects on the comprehension of the whole program. In fact, if the slice associated to an aspect would be too big (at worst the whole program), this is a hint that the separation of the aspect code from the base one is only syntactical, since in the worst case no compositional invariant can be taken for granted.

7. REFERENCES

[1] Aspectj. http://www.aspectj.org.

[2] Indus. website:http://indus.projects.cis.ksu.edu/.

[3] Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 25–29, Lancaster (UK), March 2004. Iowa State University.

[4] Lynne Blair and Mattia Monga. Reasoning on AspectJ programmes. In *Proceedings of Workshop on Aspect-Oriented Software Development*, pages 45–50, Essen, Germany, March 2003. German Informatics Society.

[5] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy,. Technical Report TR03-01a, Iowa State University, January 2003. presented at SPLAT 2003.

[6] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

```
public class T{
  int temperature;
  public T(){
   this.temperature = 0;
  }

  public void set_temp(int t){
   System.out.println("Setting temperature to "+t);
   this.temperature = t;
  }

  public void shutdown(){
   System.out.println("Shutting down...");
  }
}

public class Main {
  public void method1(T t, int x){
   t.set_temp(x);
  }

  public static void main(String[] argc){
   Main m = new Main();
   T t = new T();
   m.method1(t, Integer.parseInt(argc[0]));
  }
}
```

```
public aspect LockAspect {
  public void T.get_lock(){
   System.out.println("Lock acquired");
  }
  public void T.release_lock(){
   System.out.println("Lock released");
  }
  before(T t): target(t) && (call(void set_temp(int))){
   t.get_lock();
  }
  after(T t): target(t) && (call(void set_temp(int))){
   t.release_lock();
  }
  before(T t): target(t) && (call(void shutdown())){
   t.get_lock();
  }
  after(T t): target(t) && (call(void shutdown())){
   t.release_lock();
  }
}

public aspect TInvariant {
  before(T t, int newval):
   set(int T.temperature) && args(newval) && target(t){
    if (newval > 100) t.shutdown();
   }
}
```

Figure 4: Example of interacting aspects

[7] Remi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse, and interaction analisys of stateful aspects. In *Proceedings of the 3rd international conference of aspect-oriented software development*, Lancaster, UK, March 2004. ACM.

[8] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[9] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Gary T. Leavens, Ralf Lämmel, and Curtis Clifton, editors, *Foundations of Aspect-Oriented Languages*, March 2004.

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

[11] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. Static slicing of Java programs.

[12] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of SIGSOFT'04/FSE-12*, Newport Beach, CA, USA, November 2004. ACM.

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

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

[15] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[16] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of SIGSOFT'04/FSE-12*, pages 147–158, Newport Beach, CA, USA, 2004. ACM.

[17] Christoph Steindl. *Slicing for Object-Oriented programming languages*. PhD thesis, Johannes Kepler University Linz, 1999.

[18] Maximilian Stoerzer. Analysis of AspectJ programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, March 2003.

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

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

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

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