Modular Aspect Verification for Safer Aspect-Based Evolution

Nathan Weston, Francois Taiani, Awais Rashid

Computing Department, InfoLab21, Lancaster University, UK.
{westonn,f.taiani,marash}@comp.lancs.ac.uk

Abstract. A long-term research goal for Aspect-Oriented Programming is the modular verification of aspects such that safe evolution and reuse is facilitated. However, one of the fundamental problems with verifying aspect-oriented programs is the inability to determine the effect of the weaving process on the control flow of the program, and thus on the state of the system and subsequently the properties that hold or are introduced. We propose a novel approach to modular verification of aspect-oriented systems using aspect tagging and Data Flow analysis of Control Flow Graphs.

1 Introduction

The increasing adoption of Aspect Oriented Programming (AOP) has considerably improved the evolvability of cross-cutting concerns (monitoring, security, replication, distribution) in complex software platforms.

The power of AOP essentially lies in its ability to impact a very large code base at run-time with only one aspect. Because of this power, however, it can be extremely difficult to predict the effect of an aspect on a base program, a particularly critical issue when AOP is to be used for software evolution. How can we be sure that an aspect achieves what it is meant to? How can we prove that it does not violate properties of the base program that must be preserved? How can we verify that it does not interfere with properties other aspects are trying to introduce?

The evolution of cross-cutting concerns would benefit enormously from well-developed formal techniques to answer these questions. Ideally such techniques should provide a framework with which to check AO programs at an early stage, in order to reuse and adapt aspects in a way which is formally verifiable.

Verification techniques are being developed for AO systems, but they still lag far behind what as been achieved for the static analysis of procedural and object-oriented programs. Our intuition is that, with the proper abstractions, existing aspect-free approaches (intra- and inter-procedural analysis, points-to analysis, abstract interpretation) can be specifically adapted to AO programs to meet their particular requirements. In this paper we discuss the properties such an "aspect aware" verification approach should have to be suitable for program evolution (Section 2). Current work is presented in Section 3. We then present how the ideal could be realised in the particular case of data flow analysis using a technique we have termed “aspect tagging” (Section 4). Section 5 concludes the paper.

2 Problem Statement

Any AO program consisting of a base P and a woven aspect a can be represented by an equivalent standalone “aspect-free” program Q, on which traditional static analysis can be performed. This approach, however, suffers from a number of deficiencies that make it unattractive for aspect based software evolution. Firstly, it is very difficult to trace results obtained on Q back to the original aspect-oriented program $P + a$. Secondly, no general statement on the properties of a can be made, except in conjunction with a specific base program. This requires the whole analysis to be repeated for each base program on which a is applied. This limitation puts particular constraints on any evolution process based on program families. Thirdly, in decoupling the analysis from the AO structure of the original code, such an approach effectively bars any optimisation based on the AO nature of the program.

To circumvent those deficiencies we think that an *aspect-aware* verification approach should have the following desirable characteristics:

Modularity We feel that the naïve approach outlined above (performing analysis on the woven bytecode, thus determining whether $P + a = Q$ in terms of the properties that need to be maintained) neglects one of the key features of AOP - that is a *modular* framework, and thus requires *modular* analysis techniques. The criteria of modularity can be further broken into two sub-criteria:

- **Comprehensibility** The results we obtain using the analysis should be able to be *back-tracked* to the original program structure - that is, understandable using the terms of the encapsulation which the original program structure afforded. In real terms, this means that we will be able to see how the aspect itself has affected the properties of the system as a whole, not just how the system behaves.
- **Reuse** The results should be encapsulated in the same dimension as the aspect - that is, if the aspect is used with a different base program, the results should be able to be (at least partially) reused. This is a stronger property than comprehensibility.

Efficiency/Scalability The usefulness of formal methods for checking of safety properties is proportional to the efficiency with which they can be applied. Therefore, any analysis we can perform must have the ability to be applied within a reasonable time-frame, defined partially by the cost of failure of the system - that is, how safety-critical it is. The analysis must also be scalable - an analysis which is only applicable to trivial programs is fairly pointless. We would expect such an ideal analysis to scale to large industry-grade programs with multiple interacting aspects, as well as dynamic approaches.

Portability A desirable property of the analysis is the ability to be adapted to different languages, approaches and architectures, to maximise its usefulness.

One of the major challenges facing formal methods with AOP is determining a proper *abstraction* of the code such that program verification which fulfils the criteria above can be performed. This abstraction needs to be both *correct* - that is, encompass all the executions of the system that make sense or that we want to check - and *feasible* - that is, not containing so many possible states that state space explosion occurs and checking becomes unreasonable or useless.

This task becomes particularly hard in the presence of AOP's dynamic features, such that it can become infeasibly expensive to determine the execution of a AO system before run-time. For example, dynamic aspects could be woven at run-time; the behaviour of compile-time woven aspects could be affected by dynamic parameters; or dynamic joinpoints such as AspectJ's `cflow` could be used. An extreme total abstraction could then be that every potential aspect advice is applied at every potential joinpoint. This would clearly produce an absurd and useless abstraction, which would most likely be unable to positively determine any properties of the system.

Clearly, what is required is an abstraction of the system which is accurate enough to be *sound* - that is, proving something is true (or not) of the abstraction means that it is true (or not) of the actual system - yet useful enough to be as *complete* as we need - that is, able to give a definite answer to a proposition. If the abstraction is sound but not complete, we allow answers of "maybe" for every question we ask of the system, which is technically correct but not very helpful[1]. On the other hand, we would not want a system which gave us a definite answer for the abstraction which was not correct for the actual system. Thus, our abstraction must be correct, at least for the properties we want to check, and sound for the analysis we wish to perform.

We also require that our abstraction be modular - that is, that it retains the program structure of the actual system. We say this because we want the results of our analysis to be reused along with the aspect - recalling our example, we want our programmer to be able to get the aspect from the library and use verification techniques to determine that it will, indeed, work with his system. For this to work, it would be immensely helpful to have some result already present in the system in order to reduce computation time and effort.

Hence, we require a partial abstraction of the system, which provides us with an estimated set of potential executions which is as close as possible to the true set. Determining this abstraction is a matter of applying effective program analysis techniques - correct data-flow analysis combined with constraint-based analysis - to enable abstract interpretation [14, 7]. As we will show in Section 4, these techniques do not scale well to AO systems and require adaptation.

[1] Similarly, abstractions are often only sound for a particular class of answers - for example, if the abstraction answers "yes" for an analysis, we know the answer is "yes" for the original system; but if it answers "no" we cannot be sure. This strongly affects our choice of abstraction.

3 Current Work

There have been several notable efforts in the field of applying program analysis techniques to AOP. While all these efforts take slightly different approaches, the end goal is broadly similar - *modular* verification of aspects. The ideal goal is a complete proof that states that for every possible base system on which an aspect can be woven, and for every possible weaving within that system, the aspect will always:

1. Maintain desired properties of the base system such that the augmented (woven) system has the same properties as the original
2. Introduce its own properties to the augmented system correctly
3. Maintain desired properties that other aspects introduce

This goal is still a long way off for program analysis and, as such, most approaches seek to restrict the problem in some way.

We will divide discussion in this area into two sections - *static code analysis* techniques and *other approaches*.

3.1 Static Analysis

Recent static analysis techniques have related closely to the *categorisation* of aspects. An early attempt at this was suggested by Katz and Gil [11], in which three broad categories were proposed:

Spectative. These are simply monitoring aspects whose function is to record the actions of the base system without affecting them whatsoever.

Regulative. These aspects do not change the actions or basic functionality of the underlying system, but are often used to determine control flow in the system - an example being a contract enforcement aspect which decides whether a method is called based on pre-conditions.

Invasive. These aspects actively change the functionality or state of the underlying system in various ways. In powerful AOP systems, aspects can modify the values of both class and instance attributes or introduce their own, call methods before and after the advised joinpoint or even skip the joinpoint code completely (as is the case in an AspectJ advice with no `proceed()` statement).

Two other works also propose a classification system. Clifton and Leavens [5] suggest *observers/spectators* and *assistants* - similar to spectative and invasive aspects - and propose an extension to aspect languages by which the base object includes explicit references to the aspects which observe or assist it, enabling a more modular reasoning. Similarly, Rinard et al [16] propose a finer-grained categorisation coupled with a more powerful analysis to automatically classify interaction between advices and methods. Their work adapts an existing object oriented analysis to aspect oriented programs.

Work from Sereni and de Moor [17] proposes a reduced pointcut model based on regular expressions and use a meet-over-all-paths analysis which produces an optimised way of joinpoint matching. Although the primary goal of this work is optimisation, they acknowledge that the work could be used to determine aspect interaction - that is, when two or more different pieces of advice may be executed at the same joinpoint.

3.2 Other Approaches

Two other approaches [19, 21] encapsulate model checking assertions (using Bandera [6] and Jpf [15] respectively) within aspects, thus achieving some level of modularity. However, the actual checking then occurs on the augmented (woven) system, which prevents (partial) verification results to be attached to aspects for reuse on other base programs.

Krishnamurthi et al. [13] attempt a more modular model checking [10] technique whereby the final finite-state machine (FSM) of the woven system is constructed from the code before the aspects are woven - that is, an estimation of the final behaviour of the system is created. This uses a sophisticated backward flow analysis to determine the location of joinpoints and inserts calls to the FSM of the advice which would apply at the point. However, this approach only works for aspects which are guaranteed to return the system to the state in which the advice was called - in the terminology of [19], *spectative* aspects - which turn out to be a remarkably small subset of possible aspects. This technique can also only determine whether properties of the original base system are not violated, not whether the aspect introduces its own properties properly or affects the behaviour of other aspects.

Finally, there has been some work on formally identifying and resolving conflict or interaction between aspects. Sihman and Katz [18] develop a calculus for their superimposition system [20] which defines a general methodology for calculating how superimpositions are composed together before they are applied to the underlying system based on their specifications. However, this has yet to be implemented in a concrete AOP language.

Similarly, Douence et al [8, 9] develop an abstract formal semantics for aspects which includes rules for composition based on precedence. They demonstrate how defining composition with an order results in different behaviour which can be formally specified and hence provides a possible basis for analysis. They also propose rules for the detection of interaction. Again, this provides a very strong semantic base, but as yet is unimplemented.

The analysis system proposed by Rinard et al [16] has potential for detecting interference between aspects. In general, static analysis techniques such as program slicing [2, 3] have application in this field, although so far this has received little exploration.

In summary, the range of formal program analysis techniques currently under development for AOP systems is widening, reflecting the increasing confidence in both AOSD and formal methods. However, in the early years of the AO paradigm, program analysis techniques are generally at an early level, tend to be application-dependent at least in their implementation, or reduce the problem somewhat by considering a subset of AOP features.

In particular, flow analysis techniques that are so far developed rely on representing the aspect-oriented program in such a way that existing (object-oriented) data and control flow analysis can be applied. This necessarily means that, at the flow analysis level, we end up treating the base program and aspects as a combined, woven, object-oriented system. Even when the program is represented in a graph with the distinction between base and aspect emphasised, as in [23, 24], the analysis then occurs on the complete program. This means that *modular* analysis of the aspect's behaviour independently of a base is restricted. It is this restriction that we aim to address in our work, in the development of a modular aspect-oriented data and control flow analysis.

4 Data Flow Analysis of Aspect-Oriented Programs

4.1 Summary of Proposed Approach

Our approach can be summarised as follows:

1. Obtain the bytecode of base and aspect;
2. Classify the aspect with respect to the base;
3. Create abstract control-flow graphs of both base and aspect;
4. Tag the CFG of the aspect;
5. Create a graph transformation of the base using the CFG of the aspect;
6. Use the resulting model to create an abstraction of the augmented system using data-flow analysis.

To implement this approach two main technical goals must be achieved:

Tagging The first goal is the ability to reason about an aspect and a base such that they remain distinct in our analysis. We achieve this by the process in which we construct the augmented CFG - that is, the CFG which represents possible executions of a woven base program and aspect - by tagging the nodes of the aspect advice and using these tags in the control flow analysis we perform.

Data Flow Analysis (DFA) The second goal is the data-flow analysis of the augmented CFG. The realisation of the first goal ensures that this analysis is modular, as the effects of the aspect can be clearly seen and backtracked to the original structure via the tags we have introduced. The transformation of the CFG enables us to map existing techniques to aspect-oriented programs.

An initial difficulty is finding the location of joinpoints at which the aspect advice might be applied in our system at pre-weave time. Different AOP models use a variety of *pointcut descriptors*(PCDs) at which advice can applied, some of which are more difficult to statically determine than others. At this stage we use a simple PCD model based on pattern-matching of method signatures, with the aim of extending the model as the approach is developed, perhaps using abstract interpretation[7] for control-flow based PCDs.

From this, we extract control flow graphs from the bytecode of the base program and the aspect (extracted from the AspectJ compiler[1]). We then *tag* each node of the aspect's CFG to show us that it is part of the aspect and not the base. This is represented in Fig. 1 by means of a dashed box. When the CFGs are composed to form a model of the augmented system, the tags are maintained and give us the basis for a modular reasoning framework.

We then construct an augmented CFG by adding transitions from the joinpoints to the aspect's CFG, using an extension of the currently available Soot methods for doing so. This is comparable to existing techniques used for inter-procedural analysis, and so we transform the CFG in such a way that these traditional approaches can be used. One difficulty in the CFG transformation is the problem of aspect pointcuts which have formal parameters that need to be bound. We envisage this being equivalent to inserting a decision node based on the predicates of the joinpoint with a "method call" to the advice node if the predicates evaluate to true.[2]

For example, for simple advices, we can simply add a transition from each node corresponding to a pointcut at which the aspect applies to the beginning of the CFG of the aspect's advice, and a similar return transition, depending on what kind of advice is being applied (see Fig. 1 for an example).

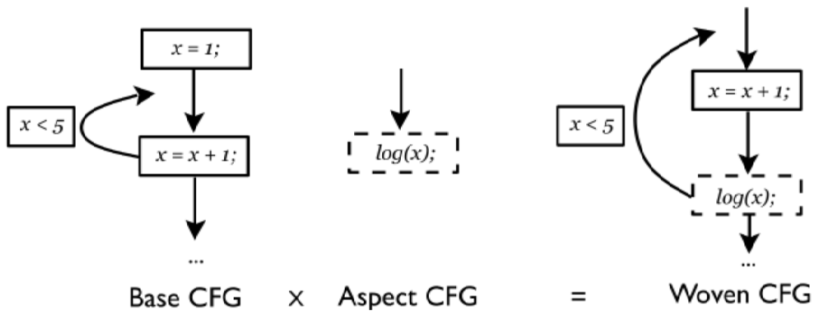

Fig. 1. The result of weaving a logging aspect on a base program consisting of an advised while loop

After this, we are left with an abstract augmented CFG on which we can perform data flow analysis.

[2] At this stage we only consider homogeneous aspects, i.e. aspects consisting of one advice relating to one concern. Heterogeneous aspects will be considered later in the development of our approach.

Here we use the classification of an aspect[16, 5, 20, 12] to determine what analysis to perform. For example, if the aspect is spectative[20] (that is, does not affect the state of the base system - e.g. a logging aspect), we do not need to check for violation of properties in the base system at all, reducing the intensiveness of the analysis. The ability to cut out stages of the analysis also enables us to reduce the level of abstraction we need to perform, meaning that we have a higher probability of obtaining meaningful results.

4.2 Adapting a Simple Analysis

To illustrate this, we show how we would attempt to adapt a simple data-flow analysis to a program in the presence of aspects. *Live variables analysis* is a classical data-flow analysis which aims to determine whether there exists, at a program point p, a path from the exit of p to a use of a variable such that there are no points on the path which redefines the variable [14]. That is to say, it aims to compute, for a given program point p, which of the variables currently defined at p can still have an impact on remaining execution of the program (i.e. are still "alive").

It is a backward flow analysis, and uses two flow sets gen_{LV}, the set of variables which appear in a block; and $kill_{LV}$, the set of variables which are killed (that is, redefined) in a block. The two flow functions $LV_{exit}(l)$ and $LV_{entry}(l)$ then calculate which variables are live at, respectively, the exit and entry of a program block labelled l. They are defined thus:

$$\begin{aligned} LV_{exit}(l) &= \begin{cases} \emptyset & if\ l \in final(S_\star) \\ \bigcup\{LV_{entry}(l') \mid (l', l) \in flow^R(S_\star)\} & otherwise \end{cases} \\ LV_{entry}(l) &= (LV_{exit}(l) \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l) \\ & \quad where B^l \in blocks(S_\star) \end{aligned}$$

where $S_\star$ is the program we are analysing; $(l', l) \in flow^R(S_\star)$ means that the program flows forwards from l to l' and thus backwards from l' to l; and B^l is the program block in $S_\star$ which has the label l. Intuitively, then, the set of equations says that, at the exit to a block, the set of live variables is exactly the set of live variables at the entry of the block following it; and at the entry to a block, the set of live variables is the set of live variables at the exit, minus those variables that have been killed (i.e. redefined) in the block, plus those variables that have been used in the block.

This analysis works well for simple programming languages without functions or procedures, that is, *intraprocedural analysis* which only operates within a single control flow. *Interprocedural analysis* [14] - that is, analysis which takes into account control being passed to other procedures, functions and advices - introduces concepts such as call and return labelling and parameter passing for procedural languages, and there has been significant work on adapting this for object-oriented languages already, e.g. [4]. Further adaptation to more complex languages requires significantly more sophisticated techniques for determining control flow, parameter passing and dynamic features of the language.

To adapt this analysis to be a) applicable to aspect-oriented languages and b) modular, we introduce the notion of *tagged flow sets*. The idea is that we encapsulate the data-flow information which is provided by the aspect in separate flow sets such that we can perform intraprocedural analysis on the aspect code, while retaining the ability to perform interprocedural analysis on the whole program. In other words, we can see how the whole program's properties are affected by the introduction of an aspect by considering the whole program. However, we can also see how an aspect would affect a certain base program given certain values for the binding of its abstract entities. We can then use this information to begin to extrapolate how the aspect would behave given certain *classes* of values - for example, whether a field is bound to a positive or a negative number - and thus create abstractions of how the aspect will affect a system, and thus create partial results which can be reused.

We introduce a set *advices*, which is the advices α which apply at a certain join point[3] ($JP_\star$) in the program $S_\star$. The advices applicable at a certain block (program point) are given by the function:

$$advices : Blocks_\star \to \mathcal{P}(Adv \times JP_\star)$$

$$where\ advices(b \in Blocks_\star) = \bigcap \{(\alpha, j) \in Adv \times JP_\star |\ b\ matches\ j\}$$

With this framework, we can reformulate the classical live variables analysis with tagged flow sets gen^A_{LV} and $kill^A_{LV}$ for an aspect A which introduces a `before()` advice α (this would be slightly different for different kinds of advice). LV_{exit} remains the same (as $flow^R$ will include the aspect statements as well as the base code), but LV_{entry} now has two forms:

$$LV_{entry}(l \in advices(S_\star)) = (LV_{exit}(l) \backslash kill^A_{LV}(B^l)) \cup gen^A_{LV}(B^l) \qquad (1)$$

$$LV_{entry}(l \notin advices) = \begin{array}{l} ((LV_{exit}(l) \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l)) \\ \cap (\bigcup \{LV_{entry}(l') \mid l' \in advices(l)\}) \end{array} \qquad (2)$$

So we now have two equations for computing live variables - one for when were dealing with a block of code thats in some aspect advice (equation (1)), and one when it isnt (equation (2)). When we are dealing with advice code, we have the same equation as previously, except using the tagged flow sets. When the code is in the base system, we compute the same as before, but we have to add in the information from the advice - so we also work out which variables have been killed from the advices which apply at that program point.

Intuitively, then, we formulate the live variables analysis based, not only on the proceeding statements in the base program, but also in the statements

[3] Here we assume static joinpoints. For dynamic joinpoints, we would have to consider the various joinpoint *shadows* - that is, the static code points at which dynamic aspects *could* apply.

contained within the code of the aspects which apply at the program point in question. Thus, we retain the encapsulation required, while still having the ability to evaluate the whole program as a single entity.

We plan to extend the Soot framework[22] to implement our approach. One of the benefits of this sophisticated optimisation framework is the ability to transform Java bytecode into an intermediate representation called Jimple, on which inspection and analysis can be performed.

4.3 Future Work

The modular verification, as described above, of a concrete aspect statically woven in a concrete base system is an appreciably difficult task which we hope our approach goes some way to resolve. However, the verification of *generic* aspects and bases is more difficult still - given an aspect with a abstract advice and an undefined joinpoint, can properties be verified? Conversely, can concrete aspect be subject to formal analysis even without a concrete base on which to weave?

We envisage that our approach can be used to facilitate more modular reasoning about the effect of generic aspects on an arbitrary base program, a future goal for our approach. Given the Soot framework's ability to generate classfiles from scratch, we may be able to produce a skeleton base program (or *dummy* program[19]) on which the weaving of a concrete aspect can be checked. Again, we hope to able to use the categorisation of the aspect to restrict the set of possible programs and/or program executions on which the weaving of the aspect makes sense, to reduce the resource intensiveness of this approach.

Especially, we envisage an application in the extremely difficult discipline of verifying dynamic AOP systems - that is, systems on which aspects can be woven, changed or removed while the program is running. Being able to produce partial results about the weaving of an aspect before it is due to be weaved would be a significant step forward in the goal of effective and verifiable reuse and evolution of dynamic Aspect-Oriented Programs.

5 Conclusion

We have presented a novel approach to the verification of aspects based on control flow analysis, using tagging to keep the base and the aspect distinct in our analysis such that the results can be backtracked to the original program structure. We envisage that bringing structural knowledge to the complex action of flow analysis will enable much more efficient static reasoning of aspect-oriented programs, and we hope to be able to map existing flow analysis techniques to analysis of such programs. We have shown possible extensions in the fields of verifying abstract aspects on abitrary base systems and verifying dynamic AOP systems.

References

1. AspectJ. Home page of the aspectj project. http://eclipse.org/aspectj.
2. Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In *In Proceedings of Foundations of Aspect-Oriented Languages Workshop 2004*, 2004.
3. Davide Balzarotti and Mattia Monga. Slicing aspectj woven code. In *In Proceedings on Foundations of Aspect-Oriented Languages Workshop 2004*, 2005.
4. Ramkrishna Chatterjee. *Modular Data-Flow Analysis of Statically Typed Object-Oriented Programming Languages*. PhD thesis, Graduate School, New Brunswick Rutgers, The State University of New Jersey, 2000.
5. Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report 02-04a, Iowa State University, Department of Computer Science, April 2002.
6. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawm Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 2000 International Conference on Software Engineering*, 2000.
7. Patrick Cousot. Abstract interpretation. Technical report, LIENS, 1996.
8. Remi Douence, Pascal Fradet, and Mario Sudholt. Detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, 2002.
9. Remi Douence, Pascal Fradet, and Mario Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, 2004.
10. Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
11. Shmuel Katz and Joseph Gil. Aspects and superimpositions. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 308–309, London, UK, 1999. Springer-Verlag.
12. Jorg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Engineering*, 2004.
13. Shiram Krishnamurthi, Kathi Fisher, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of the ACM SIGSOFT*, 2004.
14. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2nd edition, 2005.
15. Java PathFinder. Home page of the jpf project. http://javapathfinder.sourceforge.net.
16. Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, 2004.
17. Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, 2003.
18. Marcelo Sihman and Shmuel Katz. A calculus of superimpositions for distributed systems. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Engineering*, 2002.
19. Marcelo Sihman and Shmuel Katz. Model checking applications of aspects and superimpositions. In *Proceedings of the 2003 Workshop on Foundations of Aspect-Oriented Languages*, 2003.

20. Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The British Computer Society Computer Journal*, 46(5), 2003.
21. Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002.
22. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
23. Jianjun Zhao. Slicing aspect-oriented software. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 251, Washington, DC, USA, 2002. IEEE Computer Society.
24. Jianjun Zhao and Martin Rinard. System dependence graph construction for aspect-oriented programs. Technical Report MIT-LCS-TR-891, Massachusetts Institute of Technology, 2003.