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). We then present how this could be realised in the particular case of data flow analysis using a technique we have termed "aspect tagging" (Section 3). Section 4 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 It should exploit the encapsulation provided by a to produce verification results which can be reused along with the aspect.
Scalability/Efficiency Any industry-grade program should be able to be checked in a reasonable time frame.
Comprehensibility The results of the verification should be understandable in terms of the original AO program structure.

The approaches that have currently been developed do not yet fulfil these criteria. Two of these [7, 10] encapsulate model checking assertions 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 (modularity).

A more powerful analysis has been proposed by Rinard et al. in [6] to automatically classify interaction between advices and methods. Their work is very similar to what we intend to do in that it adapts an existing object oriented analysis to aspect oriented programs. However, their approach is not modular, as aspect and base program must be considered at the same time. It also does not offer a general methodology by which a standard OO analysis can be transformed into an AOP analysis while retaining aspect traceability. These are two issues we particularly aim to address in our work, as explained in the following section.

3 Data Flow Analysis of Aspect-Oriented Programs

3.1 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 check properties 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[3] 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. 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 interprocedural 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.[1]

[1] 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.

After this, we are left with an abstract augmented CFG on which we can perform data flow analysis. It has been shown[9] that such analysis can be analogised to model checking[4], as temporal logic properties expressed in CTL can be transformed to statements in terms of data flow analysis.

Because we have tagged the nodes which come from the aspect advice, we can reason about the effect of the aspect on the properties of the system in a modular fashion. We extend traditional data flow analysis with *tagged flow sets*, which apply only for the aspect advice, enabling intra-procedural analysis on the aspect while retaining the ability to perform inter-procedural analysis on the whole program. Retaining the separation of base and aspect code introduces *traceability* to our approach, enabling us to backtrack our results to the original program structure, thus laying the foundations for a truly modular verification framework.

Here we use the classification of an aspect[6, 2, 8, 5] to determine what analysis to perform. For example, if the aspect is spectative[8] (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.

We plan to extend the Soot framework[11] 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.

3.2 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[7]) 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.

4 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. 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.
3. Patrick Cousot. Abstract interpretation. Technical report, LIENS, 1996.
4. Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
5. 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.
6. 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.
7. Marcelo Sihman and Shmuel Katz. Model checking applications of aspects and superimpositions. In *Proceedings of the 2003 Conference on Foundations of Aspect-Oriented Languages*, 2003.
8. Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The British Computer Society Computer Journal*, 46(5), 2003.
9. Bernhard Steffen. Data flow analysis as model checking. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 346–365, London, UK, 1991. Springer-Verlag.
10. Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002.
11. 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.