# Improving a Modular Verification Technique for Aspect Oriented Programming

Alfons Laarman
Faculty of EEMCS
University of Twente
a.w.laarman@student.utwente.nl

## ABSTRACT

As aspect oriented software becomes more popular, there will be more demand for a method of verifying the correctness of the programs. This paper tries to address the verification issue by improving a modular verification technique proposed by Krishnamurthi et al. The technique has the problem that it can not handle every aspect, which may result in a false awnser. By checking the type of the aspect in advance, we can prevent this behavior. The proposed solution also improves some other issues regarding the model checker.

## Keywords

Incremental verification, modular verification, model checking, aspect-oriented programming, feature oriented software, aspect classification.

## 1. INTRODUCTION

Aspect oriented programming (AOP) is a new programming paradigm which deals with crosscutting concerns. These concerns are usually scattered all over the program code, while realizing the same task. A good example is logging: at every place where the programmer wants to log the status of the program, a logging function has to be called.

The goal of AOP is to separate these crosscutting concerns and place them in a separate module, called an aspect. The functionality of the aspect is imposed on the program by the definition of pointcut designators (PCDs). These PCDs represent points in the program where the code of the aspect, called advice code, needs to be applied. These individual points are also called joinpoints, because these are the places where the code of the aspect and the program itself will be unified by the compiler. For a complete introduction into AOP, see [BH05].

This type of programming allows for better modularity in the program but also poses new problems, one of which being the increased difficulty of verifiability. As AOP gradually becomes more popular, this problem will become a more important one to solve. Some work already exists that proposes solutions for model checking of AOP. This paper is an effort to change the situation by improving an existing technique for AOP model checking. In the next section, we will show the chosen technique to be a promising one. After an explanation of how the technique works, we will identify its shortcomings. The most important one is that it may return a false answer for certain types of input (aspects). The proposed solution to this shortcoming greatly improves the use of the technique in practice, because model checking is mostly used for mission critical systems.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission.
*5th Twente Student Conference on IT , Enschede 26th June, 2006*
Copyright 2006, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

## 2. RELATED WORK

In [UT02] and [DM01] a simple approach to verify AOP is explained. The properties that can be verified may be expressed in CTL. In these works, the model of the program is a composition of a model of the main program and the model of the advice code. A copy of the advice model is inlined into the model of the main program, wherever it applies. This is a static approach, which logically has the problem that the model has to be adapted for every small change in the program or in the advice code.

[NC01] takes the approach of defining a formal language, which supports concern level compositions. Even though it seems like a promising way, it is still in its early stages of development. And there is no work to make the technique useable in practice.

In [KFG04] a modular verification method is discussed. In addition, this work uses CTL to express properties. The basic idea of this paper is to use the verification of the base program to verify aspect individually. This is realized by saving the state of the verification in interfaces for every joinpoint of the base program (The left part of figure 1). So interfaces represent the verification state of the base program and consist of nothing more than properties. Any aspect that is made for a certain joinpoint can be individually verified against the interface of the joinpoint (The right part of figure 1). When verification succeeds, it can be concluded that the base program still operates correctly when the aspect is applied.

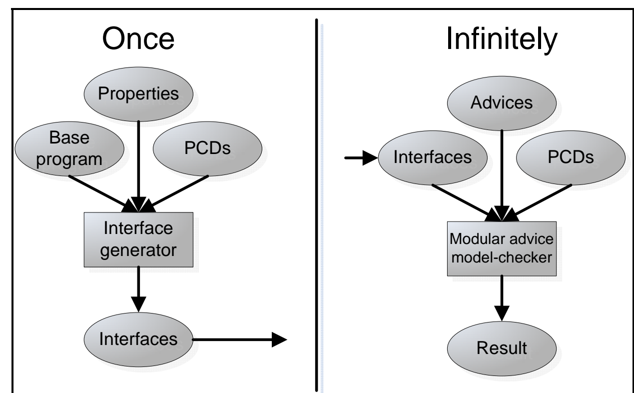


**Figure 1: The dataflow of the modular model checker**

The technique seems very promising. As Krishnamurthi et al. points out, it may aid aspect development:

1. The advice may be authored at a different time or in a different place from the program, just as modules are developed in spatial and temporal independence.

2. The advice may be edited repeatedly; verification time is proportional to the size of the system, so constantly verifying the changing advice against a fixed program is inefficient.

So a clear advantage of the technique is that it does not require simultaneous design of the advice for every PCD. Moreover it is merely an extension to verification techniques that are already successfully used in practice for non-AOP programs, like the one described in [CES86] and [CDH00].

# 3. BACKGROUND

The previous section showed in what ways the modular aspect model checker is superior to other techniques for aspect model checking. Therefore, we will focus on this technique. To be able to explain where its weaknesses lie and what can be done about them, we will first explore the technique in more detail.

According to standard model checking practices the base program is first checked, this process is henceforth called the "base verification". It must satisfy the properties, but that is not the only use of the verification process. Interfaces must be constructed at every joinpoint where advice can apply. The interfaces are the recorded state of the verification at each identified joinpoint plus some extra information about the joinpoint detection. The latter is used to detect newly created jointpoints in advices and is not of any interest for the improvements discussed here.

It is intuitively clear that the state of the verification process can be used to verify advices in isolation, because this would be the same as verifying the advice woven into the base code. However, some subtleties arise by doing so. These are discussed in section 6 of [KFG04] and will be further explored in the next section.

# 4. PROBLEM STATEMENT

## 4.1 Problems

Modular verification looks like a promising way to verify AOP, mainly because it is powerful; there is no need to repeat analysis on the base program for new aspects and aspects can be developed separately from the base program. However there are some shortcomings which are not explored enough in [KFG04] (or in others) to conclude that they cannot be eliminated. To quote Krishnamurthi et al.:

1. Around advices (Section 6). "Depending on whether *proceed* is called or not, the body of a function *f* will be executed or not. The body of *f* has already been traversed by the model checker (in the base verification), it is tempting to reuse the verification effort by adopting the labels already in the program and avoiding re-verification of the body of *f*. Reusing the labels on this copy of *f* is, unfortunately, not necessarily sound. The fragment of the advice that appears after resumption may invalidate some of the labels that are on the states of *f*. For this reason, we currently inline a copy of *f* and verify *f*'s body in the context of the proceed-resume states.…"

   For a large function *f* it logically has costly consequences to inline a copy of the body of *f*.

2. Algorithm costs (Section 11). "While the underlying model checker runs in time linear in the size of the model (which can be the base program or an advice machine), in the worst case each advice must be verified once per state in the joinpoint it advises…"

An even more important shortcoming of the technique is the fact that not all advices can be verified correctly. As [KAT06] notes, but [KFG04] fails to mention, the advice may only access fields, which are local to itself. If the advice makes a modification to a field local to the program, the base verification may not be correct anymore. [KAT06] proves this and calls such advices "invasive". It is illustrated by figure 2, where the variable *A* is local to the program (left part of the figure). When the advice (middle part) is applied at PCD *before Call(f)* we get a new state diagram (right part). We can see that the first state of the program directly after the advice (the first state of the body of *f*) is a new state and the guard in the next state is invalidated.

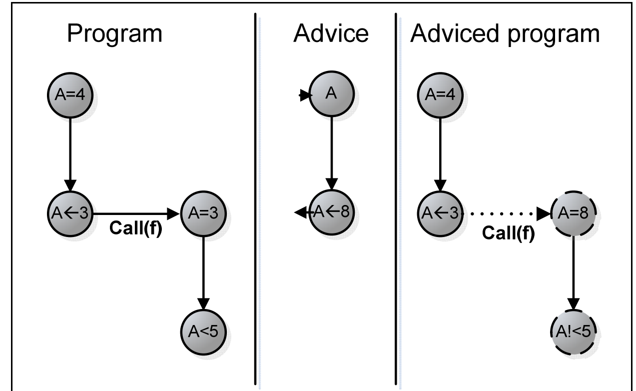


**Figure 2: program advised by an invasive aspect**

This can result in a situation where the advice is concluded to be correct, i.e. satisfying the properties of a particular interface, but the remainder of the program is now incorrect because the base verification is not sound anymore. In other words, the model checker may return a false positive.

## 4.2 Focus

To make use of the modular model checker in practice it is crucial that it operates correctly. Therefore, this paper will mainly focus on the problem of the false positives. In the next section, a practical method will be proposed and investigated to overcome this problem.

The other problems mentioned in the preceding section have a negative effect on the time complexity and applicability of the model checker.

1. Re-verification of the body of a function may be extraneous, depending on the classification of the advice. Because Krishnamurthi et al. also states about around advices: "In practice, however, we believe this will often be unnecessary. When an around advice invokes proceed, the aspect itself often performs operations orthogonal to those being advised…"

   Another problem regarding in-lining of a function is the fact that the code of the base program may not be available at the time or place of aspect verification. No precaution is taken for this case in the [KFG04].

2. Algorithm costs can be reduced by the prevention of unnecessary checking of advice for different joinpoints. In section 11, Krishnamurthi et al. propose the use of deductive techniques to pool joinpoints with similar interfaces. However, as [KAT06] notes that in many cases it is not even necessary to verify an advice at all, aspects classified as *spectative* cannot invalidate the base verification.

The next section will mainly focus on the problem of the false positives. The other problems are loosely related and can also benefit from the proposed solution, for reasons that will become clear in the subsequent sections.

# 5. Improvements

## 5.1 False positives

False positives can only be caused by invasive advice [KAT06] and the modular model checker is proven to be sound for all other advices ([KFG04] Theorems).

It would not be appropriate to try to extend the model checker so that it would be able to check invasive advices correctly. That would mean changing the fundamental approach, which makes it modular in the first place. So we can safely conclude that invasive advice is beyond the scope of this model checker.

A functional implementation of this model checker would need to be able to recognize such advice and to handle it appropriately. To recognize the classification of the advice we propose the use of the method in [RSB04], because the authors even produced a functional tool in java which is able to identify the type of any advice. This tool can be directly coupled to an implementation of the modular model checker and needs to identify the advice type as the first step of the whole verification process. Figure 3 gives an impression of the dataflow of the new model checker, the interface generator is omitted because it stays the same.

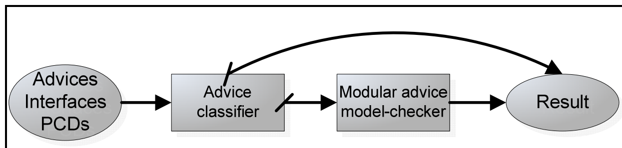


**Figure 3: The dataflow of the improved modular advice model checker**

Types of advice in [RSB04] are called classifications; Table 1 shows an overview of them. It contains two axes; the flow control axe (vertical) classifies aspects according to their influence on the execution sequence of the program and the scope axe (horizontal) classifies aspects according to the data dependencies it has with the base code.

**Table 1: A classification scheme for aspects**

| Scope / Flow control | Orthogonal | Independent | Observation | Actuation | Interference |
|---|---|---|---|---|---|
| Augmentation | | | | | |
| Narrowing | | | | | |
| Replacement | | | | | |
| Combination | | | | | |

This classification system makes finer distinction between aspects then [KAT06] does. Luckily, there are similarities between to two classification systems. [KAT06] points out the following equalities: "The latter two categories (classifications on the scope axe in [RSB04]) are invasive". The control flow, the other axe in the classification of [RSB04], does not influence the model checker as is clearly illustrated by the example Krishnamurthi et al. give for verifying *around advice*.

Until now we only explained how the model checker can be extended to identify the classification of advices and which classes should be handled appropriately. We have said nothing on how to handle them. A safe treatment would be to reject them always. However, it is possible that the advice still does not invalidate the properties we are checking; it is only our model checker, which is not able to verify it. Therefore, we may choose to reroute the input to another model checker (for example [UT02] or [DM01]) which verifies the properties on the woven program. It should be noted that we need the base code do this, so it may not be appropriate in all cases.

## 5.2 Around advice

The section "problem statement" explained the algorithm costs involved in checking around advice and the problem of the possible absence of (a model of) the base code to in-line the function in the advice.

Now that the advice classification is known in advance we can treat around advice more appropriate. [KAT06] proves that all the properties for the base program hold for spectative advice. The work also identifies the first classification on the scope axe to be definitely spectative. So whenever an aspect is classified as being orthogonal, independent or observational the model checker can refrain from in-lining a copy of the function for every proceed call.

## 5.3 Algorithm costs

For the same reason that around advice does not need to inline a copy of the function for spectative aspects, advice itself does not need to be verified whenever it is found to be of that classification. This can reduce the algorithm costs dramatically in some situations, because normally an aspect needs to be verified for every joinpoint. Further optimizations using deductive techniques as Krishnamurthi et al. propose could still be useful.

# 6. DISCUSSION

The improvements proposed here are only of some use if it does not put another burden on the already time-complex operation of model checking.

- [RSB04] does not come with conclusions regarding the time-complexity of the tool. However if we check the technologies, it was build on [BLA99] and [SAL01], we quickly see that the time-complexity is polynomial in the number of statements of the advice. This may be acceptable if the average length of advice is taken into account, which is normally much shorter than the average function of the base code. However, complex structures may have a negative effect on the pointer analysis, so it may be useful to consider some preliminary checks before the classification analysis is performed.

- Katz identifies a sub-classification of invasive aspects, called weakly-invasive: "The latter two categories are invasive, but could be further analyzed to identify special cases of weakly invasive aspects". He shows these aspects to be functioning correctly under the conditions of the modular model checker (Lemma 9). However interesting it would be to be able to identify these aspects and accept them, it is not possible to do so with the static approach of the current classification tool. To identify a weakly invasive aspect, one has to prove it returns to a state, which is already part of the base program, the latter may not be available at the time the aspects are verified. It is likely that the time-complexity of this job does exceed that of the model checker.

## 7. CONCLUSION

We have shown modular aspect model checking [KFG04] to be a promising technique to verify AOP correctness. Through its inherent shortcomings, it is not able to verify each class of advice correctly. We have proposed a way to identify the class of advices prior to model checking, so that they can be handled by a different model checker or at least be rejected all together.

To do this we make use of two works regarding aspect classification. The tool from [RSB04] can be used to identify the classification of an aspect automatically and [KAT06] delivers the proofs which classes can be handled by the model checker and which not.

The result still leaves much room for improvement; since the automatic identification tool cannot make a distinction fine enough to implement the perfect filter (weakly invasive aspects can only be identified dynamically).

## ACKNOWLEDGMENTS

Lodewijk Bergmans and Mariëlle Stoelinga helped me with the definition of concrete methods and boundaries for the proposed research. My fellow students helped me find a good research domain by sharing their findings.

## REFERENCES

[BH05] Johan Brichau and Theo D'Hondt. Introduction to Aspect-Oriented Software Development. *European Network of Excellence on Aspect-Oriented Software Development, August 2005.*

[BLA99] B. Blanchet. Escape analysis for object oriented languages. Application to Java*. In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, CO, Nov. 1999.*

[CDH00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code*. In International Conference on Software Engineering*, 2000, pages 439–448.

[CES86] Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACMTrans. Program. Lang. Syst. 8*, 2, 244–263.

[DM01] Denaro, G. and M. Monga. An experience on verification of aspect properties, *In International Workshop on Principles of Software Evolution*, September 2001

[KAT06] S. Katz. Aspect categories and classes of temporal properties*. In Transactions on Aspect Oriented Software Development, Volume 1, LNCS 3880, pages 106–134, 2006.*

[KFG04] Shriram Krishnamurthi, Kathi Fisler, Michael Greenberg. Verifying Aspect Advice Modularly, *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2004.

[NC01] Nelson, T., D. D. Cowan and P. S. C. Alencar. Supporting formal verification of crosscutting concerns. *In Reflection*, *2001, pages 153–169.*

[RSB04] M Rinard, A. Salcianu, and S. Bugrara, A classification system and analysis for aspect-oriented programs*, International Conference on Foundations of Software Engineering (FSE), 2004*

[SAL01] A. Salcianu. Pointer analysis and its applications for Java programs. MEng thesis, Massachusetts Institute of Technology, September 2001.

[UT02] Ubayashi, N. and T. Tamai. Aspect oriented programming with model checking*. In International Conference on Aspect-Oriented Software Development,* April 2002, 148–154