



Cost Efficient Dependable Electronic Systems

Formal Verification of Fault Tolerance Aspects

Author	Daniel Larsson, Ruben Alexandersson
Document Id	005
Date	15 August 2005
Availability Status	Public Final

Formal Verification of Fault Tolerance Aspects

Daniel Larsson and Ruben Alexandersson
Chalmers University of Technology
Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden
Email: {danla, ruben}@chalmers.se

I. INTRODUCTION

It would be useful to be able to use formal methods to verify the fault tolerance of software. This is however not as simple as verifying other properties. The reason for this is that it must be shown that it is the fault tolerance code that assures the correct (or acceptable) output and not the functional core code. This is a problem since the functional code always produces the correct output at verification time (except if a software fault model is assumed). And since the fault tolerance code is normally tangled together with the rest of the program code it can not be known by the verification system which parts of the code that can be used for the proof. Present work at our department is examining how aspect oriented programming (AOP) could be used for adding fault tolerance to software [2]. The separation of concerns achieved by using AOP enables us to separate the fault tolerance code from the rest of the software and should hence theoretically enable us to formally verify the fault tolerance property of a program. The goal is to be able to formally specify the fault tolerance aspects using some specification language and formally verify that the implementations of the aspects are correct with respect to the specification.

Languages like Java and AspectJ are semantically very complex and verification of programs in such languages therefore requires a substantial amount of user interaction. One approach that has shown very successful is deductive verification based on symbolic execution of the code. This is the approach used in the KeY project [1], which therefore works directly on the source code. This is a fundamental necessity for the purpose of fault tolerance verification since it is only on the source code level that the fault tolerance code is separated from the rest of the program code. Once compiled they can not be distinguished. KeY currently supports most of sequential Java, and we are presently investigating how the KeY tool could be extended to support verification of fault tolerance aspects written in AspectJ.

To our knowledge, the only specification language tailored to AspectJ existing today is Pipa [4]. Pipa is an extension of the Java Modeling Language (JML), a widely accepted behavioral interface specification language for Java. The Pipa language could likely be useful for, or adapted to suit, our needs.

II. VERIFICATION APPROACHES

For the general problem of formally verifying aspect oriented programs there are a few possible different situations and approaches. One might want to verify aspect code with respect to an existing basic program or independently of any basic program. When considering the case with an existing basic program there are two alternative approaches: either to transform the AspectJ program into a semantically equivalent Java program before trying to verify it or try to handle the AOP constructs directly in the verification system.

Let us start by discussing the situation where a basic program is available and with the approach of transforming the AspectJ program into a Java program. This is exactly what is done by the AspectJ compiler ajc, but on the byte code level. (Most other compilers for AOP languages and earlier versions of ajc perform the weaving directly on the source code.) If using this approach the aspect specifications must also be transformed in a proper way. This is what is proposed in [4], where the authors first define Pipa and then sketch how to transform an AspectJ program specified with Pipa into a Java program specified with JML. They propose to modify ajc to perform this transformation. The advantage of this is that conventional verifiers that can handle Java + JML can be used. This approach is however not possible for verifying fault tolerance properties since it breaks the separation of the fault tolerance code from the basic program code.

The situation where a basic program is available and the AOP constructs are handled directly in the verification system leads to a number of problems that need to be tackled. As has been shown in e.g. [3], modular reasoning is not possible in aspect-oriented programs. *"A language supports modular reasoning if the actions of a module M written in that language can be understood based solely on the code contained in M along with the specifications of any modules referred to by M."* [3] This is obviously not true about AspectJ since an aspect, not part of and not referred to by a class C, can change the behavior of C. This means that before an advice that is part of a fault tolerance aspect can be verified a whole-program analysis to identify all joinpoints that match the pointcut that belongs to the advice must be performed. When all joinpoints that match the advice's pointcut has been identified the deductive verification can be performed, and will produce one proof branch for each joinpoint. This will lead to potentially very large proofs. Moreover, rigorous definitions of

the new language constructs added by AOP is needed. Even though this approach might be applicable in the case of fault tolerance, since the aspect and base program code are still separated, it is not trivial to assure that the base program code is not used as part of the proof.

Let us now consider the case of verifying fault tolerance aspects *independently of any basic program*. This approach would clearly circumvent the problem of dependence of the basic program in the verification of fault tolerance code. Moreover, it would make it possible to create libraries of highly dependable and reusable fault tolerance mechanisms in form of aspects. This is possible since dependencies from the fault tolerance aspect on the base program for producing acceptable output does in itself disprove fault tolerance. To our knowledge this has never been done and the technique might, if further developed, become useful for building dependable reusable libraries of aspects assuring other properties of programs as well.

If this latest approach is chosen the advice code of the aspects must be verified without knowledge of the joinpoints that they match. When considering AspectJ there are five different kinds of advice that need to be handled: **before**, **after**, **after returning**, **after throwing**, and **around** advice. However, all these advice can be expressed using just the around advice. Moreover, it is sometimes useful to be able to reason about a collection of advice as a coherent unit; typically if they cooperate to perform a certain task. This can be done by translating this collection of advice, perhaps a before advice and an after advice, to a semantically equivalent around advice. So, it all boils down to the challenge of handling the around advice. The problem is then that an around advice may contain invocations of parts of the basic program—in form of the `proceed` statement—and since no basic program is available, nothing is known about the result of these invocations. More specifically:

- If the `proceed` call returns a value nothing more than its type is known.
- Nothing is known about what side-effects an invocation has. This means that no assertions can be made about the parts of the system state that might be modified as a result of the `proceed` call.

In order for our verification technique to be sound we have to take these consequences into account. Otherwise, we might “prove” an advice to be correct that, when applied to a certain basic program, behaves incorrectly. Here follows an overview of how this can be handled for an around advice specified with Pipa.

The Pipa language has means to split the specification of an around advice in two parts: the first covering the code before the `proceed` statement and the second covering the code after the `proceed`. (Here, for the sake of clarity, the case of just *one* `proceed` statement in the advice is considered.) The first part should be simple to verify with respect to its specification since it does not depend on the result of executing the advice’s

join points. The second part, however, might use the return value of the `proceed` statement and might make assertions about parts of the state that is modified when executing the `proceed` statement. The return value of the `proceed` call can be handled in the same way as input arguments to a method: its value is simply unknown and as soon as the deduction depends on this value a case split is made. For example, in the case of a conditional statement where the test expression depends on this return value, the proof will at this point split in two proof branches that both have to be proved. A similar approach could be used for the side-effect problem. All variables whose values might influence the correctness of the advice code with respect to the specification and that might be modified by the basic program are identified. These variables are then considered to have an unknown value after the execution of the `proceed` statement. Again, this means that when the deduction is performed and as soon as the deduction depends on the value of one of these variables a case split is made.

III. SUMMARY

The starting point is the idea of using aspect-oriented programming to add fault tolerance to software. The resulting separation of the fault tolerance code from the function code opens up the possibility of formally verifying the fault tolerance code with respect to some specification. For the general problem of verifying aspect-oriented programs there are two different approaches: the aspect code is verified with respect to an existing basic program or independently of any basic program. The first approach is in general not applicable in the case of fault tolerance aspects. The reason for this is that the correctness proof of the fault tolerance code is then dependent of the basic program’s behavior which in itself disproves fault tolerance. The second approach is however a promising one and would potentially make it possible to create libraries of highly dependable fault tolerance mechanisms that can be applied to many different applications. The essential problem to be handled is that the aspect code may contain invocations of parts of the basic program. Since no basic program is available, nothing is known about the result of these invocations. Some possible solutions to this problem is discussed.

REFERENCES

- [1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
- [2] R. Alexandersson, P. Öhman, and M. Ivarsson. Aspect oriented software implemented node level fault tolerance. In *Ninth IASTED International Conference on Software Engineering and Applications (SEA 2005)*, Phoenix, AZ, USA, November 2005.
- [3] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning, 2002.
- [4] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for aspectj. In *Fundamental Approaches to Software Engineering (FASE 2003)*, Warsaw, Poland, April 2003.