



Cost Efficient Dependable Electronic Systems

Formal Verification of Fault Tolerance Aspects

Author	Daniel Örstadius
Document Id	009
Date	9 December 2005
Availability Status	Public Final

Formal Verification of Fault Tolerance Aspects

Master's thesis

BY DANIEL ÖRSTADIUS 810122-4056

SUPERVISOR: DANIEL LARSSON
EXAMINER: REINER HÄHNLE

Göteborg, 9th December 2005

Abstract

In this Master's thesis the topic of verifying aspects implemented using aspect-oriented programming in AspectJ is analyzed. The focus is on verifying that an aspect will provide the desired functionality with no dependence on the code it advises. Of special interest is the case where aspects are used to implement fault tolerance mechanisms. The starting point is verification by theorem proving as in the KeY Prover and specification using the Java Modeling Language (JML). Solutions to handling both return values and unknown side-effects are discussed. A possible interpretation of specifications in JML, but in the context of aspects is also provided.

Sammanfattning

Det här examensarbetet analyserar problemet med att verifiera aspekter implementerade med aspekt-orienterad programmering i AspectJ. Målet är att verifiera att en aspekt bidrar med den önskade funktionaliteten utan beroende av vilket program den kombineras med. Speciellt intressant är fallet då feltoleransmekanismer implementeras med aspekter. Utgångspunkterna är verifiering genom teorembevisning så som det sker i KeY och specifikation med “The Java Modeling Language” (JML). Lösningar för att hantera både returvärden och framtida sidoeffekter diskuteras. En möjlig tolkning av specifikationer i JML, men för aspekter, ges även.

Preface

This is a Master's thesis in Computer Science at the department of Computer Science and Engineering, Chalmers University of Technology. The thesis was supervised by PhD student Daniel Larsson. The examiner was Professor Reiner Hähnle.

The work was carried out as part of the CEDES (Cost Efficient Dependable Electronic Systems) project.

Contents

1	Introduction	1
1.1	Outline	2
2	Background	3
2.1	Aspect-oriented programming	3
2.2	The Java Modeling language	4
2.3	Formal verification and the KeY Prover	5
2.4	Fault tolerance	6
2.5	Motivation	6
2.5.1	Example	7
2.5.2	Purpose of the project	8
2.5.3	Verifying aspects	8
2.5.4	Other approaches to verifying aspects	10
3	Formal specification	11
3.1	Interpretation of specifications for aspects	11
3.1.1	Non-termination	12
3.1.2	Exceptions	14
3.1.3	Conclusion	16
3.1.4	Translation to Java Dynamic Logic	17
3.2	Specification languages for AspectJ	18
3.2.1	The proceeds clause	18
3.3	Specification of fault tolerance aspects	19
4	Formal verification	21
4.1	Handling the return value of proceed	21
4.1.1	Verification using quantification	22
4.1.2	Taclet	22
4.2	Side-effects of proceed	23
4.2.1	Problem definition	23
4.2.2	Introduce universal quantifications	27
4.3	The thisJoinPoint construct	28
4.3.1	Verify for all return values	29
5	Result and discussion	30
5.1	Result	30
5.1.1	Specification	30
5.1.2	Verification	30

5.2	Discussion	31
5.2.1	Loose ends	31
5.2.2	Limitations	32
5.2.3	Is it useful?	32
5.2.4	Future work	32
5.2.5	Conclusion	33
	References	34

Chapter 1

Introduction

The goal of this thesis is to analyze the problem of applying formal methods to aspect-oriented programming (AOP) [10]. We are especially interested in the case where AOP is used to provide fault tolerance mechanisms.

Through AOP *cross-cutting concerns* can be captured in *aspects*, a construct similar to classes in object-oriented programming. Aspects can be used to implement fault tolerance mechanisms. Using aspects certain points in the execution of a program, such as calls to methods, may be captured and control flow at those points transferred to *advice*. This can for example be used to execute a method twice and compare the return values to eliminate the effects of transient faults. The main reason for using AOP is that instead of having the code implementing certain mechanisms scattered across a program, that code can be separated into aspects. For this thesis, fault tolerance is the primary example of such a mechanism. We will call the program that the aspect advises *the base program*.

AspectJ [9], presently the most widely used AOP language, is an extension of Java. In this thesis the focus is on AOP as implemented in AspectJ.

Formal specification means giving a mathematically exact description of what a program should do. With formal verification a program can be proven correct with respect to a specification through a mathematical proof. This can be done via theorem proving as in the KeY Prover, which is part of the KeY Project [1]. The KeY Prover can handle Java Card, which is a subset of the Java programming language. In this thesis we study formal verification as implemented in the KeY Prover.

We want to apply formal specification and verification to the code in aspects. That is, we want to verify only the aspects themselves, not the programs they are combined with. Concretely, it is interesting to examine how symbolic execution as used in the KeY Prover could be used to handle the constructs available in AOP. We will not deal with verification of an aspect and a base program combined into a complete program. The problem of verifying fault tolerance mechanisms is addressed in [11]. That paper presents the problem and discusses possible solutions.

We are interested in aspects that can be reused in many settings. For example, if an aspect implements a certain fault tolerance mechanism, it might be useful to be able to add the aspect to any program and know that the aspect always provides the desired property. We want to verify that an aspect

provides the same functionality as independently as possible of the behavior of the classes it advises. Not to assume anything about the behavior of the base program when verifying the aspect severely restricts the way the aspect can be programmed, as will be discussed. It is still an open question to what extent useful aspects can be constructed under these restrictions. Non the less, formal methods could allow dependable and reusable aspects to be developed.

Normally when a method is verified all the code that can be reached during its execution must be available. When verifying an independent aspect, we do not have access to the program the aspect will be applied to. This means that we have to perform verification of advice without having all the code which can be reached during its execution when it is combined with a base program. It is mainly the potential effects of the unavailable code which sets the problem apart from verifying methods in Java. Both regarding specification and verification the problems that have to be dealt with are due to that what can happen in the code of the future base program.

The solutions to this problem will concern both the way advice are specified and how the new constructs in AspectJ can be handled in verification.

1.1 Outline

Chapter 2 provides introductions to AOP, the Java Modeling Language (JML), formal verification and fault tolerance. It also further motivates the subject of verifying aspects.

Chapter 3 tries to adjust JML to be usable with aspects. It discusses problems with specifying advice as if they were methods and presents possible solutions.

Then, in chapter 4, the problems with handling AspectJ in verification are detailed. Formal verification as in the KeY Prover is the starting point.

The last chapter sums up and discusses the results.

Chapter 2

Background

In this chapter brief introductions to aspect-oriented programming, the Java Modeling Language, formal verification and fault tolerance are given. The motivation for verifying independent aspects is also presented.

2.1 Aspect-oriented programming

AOP provides means of modularizing cross-cutting concerns. Such concerns often cannot be clearly captured by other programming methodologies. Examples are security policies and logging. They are cross-cutting since they need to be addressed in more than one module of an application. With AOP a cross-cutting concern can be addressed in a single module.

Among the constructs added by AspectJ to Java are *advice*, *pointcuts*, and *aspects*. Additionally, it introduces the concept of a *join point*. A join point is a well defined point in the program flow, for example method calls or accesses to fields. Through pointcuts sets of join points may be designated. Advice are pieces of code similar to methods which can be executed before, after or around the join points picked out by pointcuts. Below is an example of an aspect.

```
aspect SimpleAspect {

    pointcut simplePointcut(int x):
        call(void someClass.someMethod(int)) && args(x);

    void around(int y): simplePointcut(y) {
        System.out.println(
            "This is printed before executing 'someMethod' ");
        System.out.println("y = " + y);
        proceed(y);
        System.out.println(
            "This is printed after executing 'someMethod' ");
    }
}
```

The aspect is called `simpleAspect`. It contains a pointcut named `somePointcut` which picks out calls to the method `someMethod` in `someClass`. Calls to

`someMethod` will be intercepted and the code in the *around* advice above will be executed instead. Inside advice a statement called `proceed` is available. It will execute the original computation of the join point, which in this case is the method `someMethod`.

The arguments of the join points can also be accessed in the advice. In this case `someMethod` has a parameter of type `int`. It is picked out by using `args(x)` in the pointcut declaration. The variable `y` in the advice refers to the argument during the execution of the advice.

Aspects can be declared as abstract and contain abstract pointcuts, which have yet to be assigned sets of join points. Since we are interested in independent aspects, it is the case of abstract pointcuts which is interesting.

In this work we primarily consider the *around* advice since the other kinds of advice can be derived from it.

We call the program that the aspect advises *the base program*. In the example above the class `someClass` is part of the base program. The process of producing a complete program from aspects and base code is known as *weaving*.

It is worth noting that many of the constructs in AspectJ are not on the level of executable statements. Since we are only concerned with verifying the code in advice, many of the language additions in AspectJ are not relevant for this thesis.

Non-executable constructs deal with among other things how pointcuts are designated, static crosscutting (introducing new fields and methods to existing classes) and different kinds of advice. The important additions on the statement level are `thisJoinPoint` and `proceed`. `thisJoinPoint` is a special variable available inside the bodies of advice, which provides information about the join point where the advice is executing.

In [5] it has been shown that AspectJ does not support *modular reasoning*. Modular reasoning means that an application can be understood one module at a time. This poses a problem when formally dealing with AspectJ. It might be, for example, that an aspect breaks the specification of a class it advises. This is a problem which will not be considered further in this thesis. In [2] there is a proposal for how to add modular reasoning to aspect-oriented languages.

2.2 The Java Modeling language

The Java Modeling language (JML) [13, 12] is a *behavioral interface specification language* (BISL) for Java. It is used in detailed design of Java modules. JML describes a module's interface and behavior. This can be done through design by contract (DBC), where the principal idea is that a class and its clients have a "contract" with each other. This contract is expressed with pre- and postconditions. If the caller fulfills the precondition when the method is called, then the class guarantees that the postcondition will hold when the method returns.

When a method is called there are three possible outcomes. It can return normally, return exceptionally or not return at all. With JML postconditions for both normal and exceptional returns as well as preconditions for when a method is allowed not to return may be specified. Specifications written in JML can be translated into Dynamic Logic (see section 2.3) by the KeY Prover.

In JML a method can be specified using the following syntax.

```

behavior
  requires A;
  ensures B;
  signals (ExceptionType) C;
  diverges D;

```

`behavior` can be replaced by `normal_behavior` or `exceptional_behavior`, but these are just syntactic sugar.

The precondition is `requires A` and `ensures B` is the postcondition for returning normally. The clause `signals (ExceptionType) C` states that the predicate `C` must be true if the method returns exceptionally by throwing an exception of type `ExceptionType`. The `diverges` clause states a precondition for when the method may not return to the caller. Unless the predicate in the `requires` clause is true nothing is guaranteed about the behavior of the method.

In the predicates of the JML clauses arguments and the fields of the class may be used. Local variables cannot appear. The construct `\result` may be used to refer to the return value of the method. JML also permits to use *model fields*, which are variables used only in the specification. Relations can be defined between model fields and the actual variables used in the implementation.

In [15] an extension of JML for AspectJ called *Pipa* is presented. In the same vein, it is in [5] discussed how to use JML and extensions of it to specify AspectJ. We use JML and these two papers as the starting point when looking at specification of advice. As will be discussed in section 3.1, it is the interpretation of the specifications when used with advice which poses the main problem.

2.3 Formal verification and the KeY Prover

The process of constructing a proof that a program obeys its specification is called formal verification. Dynamic Logic [8] is a system for reasoning formally about programs.

In Dynamic Logic the behavior of a program is expressed using pre- and postconditions. It has two notions of correctness, total and partial. Partial correctness means that whenever the precondition is satisfied, the program will satisfy the postcondition if it halts. Total correctness is partial correctness with the extra condition that the program will halt.

One method to formally verify software is through theorem proving. This is the approach taken in the KeY Project which includes an interactive and automated theorem prover called the KeY Prover. In the KeY Prover specifications are translated into an instance of Dynamic Logic. Using Dynamic Logic Java programs can be expressed and handled formally. The Dynamic Logic used to formalize Java Card is defined in [4]. The translation of a program and its specification results in a proof obligation.

How the constructs of Java are to be handled is expressed as rules in a *sequent calculus*. The sequent calculus is a deductive system for both first order logic and Java Dynamic Logic. The steps in the proof follow the program flow and the rules in the sequent calculus can be seen as performing a symbolic execution. The proof rules are implemented in the form of *taclets* [7]. Using taclets the logical content of the rules in the sequent calculus can be expressed. They can also state restrictions on where and how the rules can be applied. An example of a simple taclet is given below (from [7]).

```
find (b -> c ==>) if (b ==>) replacewith(c ==>)
```

The semantics of Dynamic Logic is *operational*. That is, the semantics of programs are defined by their transformation of the state. The state is the assignment of values to variables. To record the change of the state during the symbolic execution a construct known as *updates* is used. Updates define the states in which formulas are evaluated.

There are two kinds of variables in the Dynamic Logic for Java, *program variables* and *logical variables*. Program variables correspond to the usual notion of variables in Java. They cannot be quantified and their values can be changed from state to state by programs. Logical variables are on the other hand assigned the same values in all states and can be quantified.

As mentioned in section 2.1, AspectJ adds to Java two new constructs on the statement level, `proceed` and `thisJoinPoint`. To verify advice we need means to handle these additions during the symbolic execution. Since we are interested in independent aspects, it is unknown when verifying what calls to `proceed` and uses of `thisJoinPoint` will refer to. This means that the verification is further complicated. It will be necessary to create rules in the calculus or design other methods to handle these new constructs so that the specification of an advice can only be verified if the specification holds for all base programs. This is addressed in chapter 4.

2.4 Fault tolerance

The ability for software to detect and recover from a fault is called fault tolerance. It is a crucial property for safety-critical systems. In [3] it has been shown that aspect-oriented methodologies are suitable for implementing software fault tolerance. A number of fault tolerance mechanisms in the form of aspects are shown in the paper. By the use of AOP the fault tolerance code becomes completely separated from the functionality of the base program.

To verify the fault tolerance of software it must be shown that it is the fault tolerance code which assures the acceptable behavior [11]. This further motivates the approach taken in this thesis - that only the code in the aspects should be used in the proof. We believe that it could be useful to be able to specify and verify aspects independently of any base program.

The results of this thesis are not limited to be applied on fault tolerance aspects, but can be used with any aspect. However, fault tolerance aspects serve as the main motivation for the project. Since the nature of fault tolerance is assuring the correct behavior of software, it is desirable to apply formal methods to the mechanisms themselves.

2.5 Motivation

The following subsections motivate and give an overview of verification of independent aspects.

2.5.1 Example

In this section the use of a simple fault tolerance mechanism implemented using AOP is presented. Assume that we have the method below.

```
<someType> someMethod() {
    <someStatements>
    int[] arr = <someIntegerArray>;
    int result = doCalculations(arr);
    <someMoreStatements>
}
```

Suppose that it is necessary for the method `doCalculations(arr)` to return a value greater than or equal to zero. Even though the code in this method always returns a correct value, erroneous behavior may appear because of for example bit flips from radiation. To introduce a level of fault tolerance to the method `doCalculations(arr)`, AOP can be used. With AOP code can be inserted to run before, after or around join points in the program. To ensure that the return value of a method always is zero or greater we devise the following piece of advice with specification.

```
abstract aspect FaultTolerance {
    abstract pointcut Calculations(int[] arr);

    /*@
     * requires true;
     * ensures \result >= 0;
     */

    int around(int[] x): Calculations(x) {
        int result;
        result = proceed(x);
        if (result < 0) {result = 0;}
        return result;
    }
}
```

The pointcut is declared as abstract, which means that it has yet to be assigned a set of join points. Control flow at all the join points matching the instantiation of the pointcut `Calculations` will be transferred to the *around* advice above. Through the call to `proceed` the control flow of the program will proceed back to the join point, which will then be executed before returning to the advice. As can be seen the advice will check the return value of the join point and make sure that it is zero or greater.

This behavior is expressed in the JML specification. The advice is specified as if it was an ordinary method. As we will see, the JML clauses `signals` and `diverges` cannot be applied in a straightforward manner to advice. We will try to consider the different problems with verifying advice separately. The `signals` and `diverges` clauses are considered to be unspecified in the example above.

Before the aspect can be used it must be instantiated, as follows.

```

aspect ReturnValue extends FaultTolerance {
    pointcut Calculations(int[] arr) :
        call(int doCalculations(int[])) && args(arr);
}

```

The pointcut `Calculations` will now match any call to the method `doCalculations`. It will also expose the integer array sent as an argument to the code in the advice.

2.5.2 Purpose of the project

The goal of this project is to analyze the problem of verifying advice independently of any base program. In the example above the base program consists of the methods `someMethod` and `doCalculations`. We want to be able to verify *around* advice, such as the one above, if it will fulfill its specification regardless of the behavior of the code at the matching join points. Conversely, we don't want to be able to verify an advice if its specification can be violated depending on how the aspect is used.

This could allow libraries of reusable and dependable aspects to be built. The users of the aspects just give the abstract pointcuts concrete definitions and would not need to know about the implementation details. In the example above, it would be enough to look at the specification to see that the advice ensures that the matching join points always returns a value greater than or equal to zero.

2.5.3 Verifying aspects

There are a number of difficulties with verifying an aspect when the aspect is supposed to be independent of any base program. The source of the problems is that it is unknown at verification time what the `proceed` call will refer to. Control flow will be transferred to a piece of code which is not available.

For this work it is assumed that the aspects only have access to the base program through what is exposed by the pointcuts (although `thisJoinPoint` also could be used to modify the base program). It is also assumed that the base programs does not have access to the aspects. The verified aspects should not be visible from the base programs.

For *before* and *after* advice the base program is not able to interfere with the execution of the advice (they have no `proceed` statement). In the case of *around* advice the situation is different. The `proceed` statement transfers control flow to the base program. After the code at the advised join point has finished executing the execution returns to the advice. Since we don't have access to a base program we need to take all possible ways in which a future base program could affect the advice into account.

Since the return value (if any) from `proceed` is unknown, the proof will have to be performed for all possible values of the variable where the return value is stored. This is similar to the way input arguments to a method are treated. A possible solution to this issue is discussed in section 4.1.

Also, future base program might have side-effects affecting the state of the advice. To illustrate the problem of side-effects in the base program consider the following aspect.

```

abstract aspect Reset {
  abstract pointcut Calculations(int[] arr);

  void around(int[] x) : Calculations(x) {
    for (int i=0; i < x.length; i++)
      x[i] = 0;
    proceed(x);
  }
}

```

The purpose of the advice is to reset all the elements of an input array to zero after some calculations have been performed. This is expressed in the specification below.

```

/*@
  requires true;
  ensures \forall int i;
           0 < i && i < x.length;
           x[i] == 0;
@*/

```

But, there is a trivial error in the implementation. The advice resets the array before calling `proceed`. It will then be possible for the base program to modify the array and the advice will not reset it after execution of `proceed` has finished. It is important that we design our verification of advice so that the advice above cannot be verified. To do this we need to deal with possible side-effects in the call to `proceed`.

An attempt to handle the modifications of data in the base program could be to treat the data shared between the advice and the base program as having an arbitrary value after the `proceed` call. Such a solution is sketched in section 4.2.2.

By taking both the return value of `proceed` and potential side-effects into account when verifying we hope to achieve formal verifications of advice where there is no dependence on the behavior of a future base program for the advice to fulfill its specification.

In addition to these problems we have to deal with the possibility that, in the woven program, the calls to `proceed` may return exceptionally or not return at all. This means that the `signals` and `diverges` clauses stated in the advice can be violated by the base program. For example, if the advice uses `diverges false`, then the advice must return. But, if the verification is done without access to a base program, it cannot possibly be known whether the call to `proceed` will actually return. Hence it is not known if the advice will return in the woven program. Use of the `signals` clause causes a similar problem.

To summarize we have to deal with the following problems:

- The return value of the join point is unknown
- The execution of the base program code in the woven program might have side-effects affecting the advice
- An exception might be thrown in the base program

- The base program might not even terminate

Also, in advice the special variable `thisJoinPoint` is available to provide reflective information about the join point. We need methods to deal with these issues. In the rest of this thesis it is sketched how to formally verify an aspect taking these problems into account. This means both stating a suitable semantics for the specifications as well as designing rules for symbolic execution.

2.5.4 Other approaches to verifying aspects

Verifying an aspect separate from any base program is discussed in [6]. The authors use AOP to implement concurrency concerns and then want to prove that the aspect guarantees the absence of deadlocks independently of the base code. Model checking was used as verification technique. In this project that technique is not considered, instead the focus is on theorem proving which is used in the KeY Prover. In [14] an approach similar to the one in [6] is taken, also using model checking. Although the aim of [14] is not on verifying independently of a base program, such a solution is discussed.

In general, the advantages with using model checking is that it is both automatic and can produce counter examples. However, the number the states grows exponentially, known as the state explosion problem. Theorem proving does not suffer from that problem, but on the other hand often requires interaction. For the more specific problem of verifying aspects, it is not clear which technique is most suitable.

Chapter 3

Formal specification

This chapter contains an analysis of the problem of specifying advice. The starting point is specification in the way methods are specified using JML [13, 12]. We want to adapt JML to be usable with advice.

First we look at the problems that occur when advice are specified as if they were methods. Then existing specification languages for AspectJ are presented. Finally difficulties with specifying a particular fault tolerance aspect are discussed.

3.1 Interpretation of specifications for aspects

It seems possible to use JML with advice. Advice are very similar to methods. But when verifying independent aspects all the code which can be executed from the advice in the woven program is not available. As will be shown, this implies that the semantics of JML clauses cannot be directly applied to advice. To remedy this problem, we propose to change the interpretation of JML clauses when they are used with advice. The potential effects of the code in the future base program causes our proposed semantics of specifications to be less strict than when used with methods.

There are at least two important reasons why the usual interpretation of JML clauses is not suitable.

It may be the case that the base program, after being called with `proceed` never returns to the advice. This happens for example if the base program loops infinitely or aborts execution of the program. Problems arise when the termination of advice is specified. The proposed solution is, as will be described below, that the semantics of the `diverges` clause is changed.

The second problem concerns `signals` clauses. Suppose we specify advice using `signals`.

```
signals (ExceptionType) S;
```

Then it must be proved that if the advice returns by throwing an exception of type `ExceptionType`, `S` holds. However, it is of course unknown what exceptions will be thrown in the base program. Suppose that it can be verified that `S` holds when the advice returns exceptionally with `ExceptionType` thrown. If a future base program returns by throwing an exception of type `ExceptionType`, and this

exception is not caught in the advice, then it might not be the case that *S* is true. This will make it possible for the base program to violate the specification of the advice.

For this discussion we assume that advice are specified like this.

```
/*@
    requires A;
    ensures B;
    diverges C;
    signals (ExceptionType) S;
@*/
```

The issues of non-termination and exception throwing will be discussed independently of each other in the following two sections.

3.1.1 Non-termination

First we look at the problem of non-termination, both in the advice and in the base program. When specifying an ordinary method in JML the `diverges` clause can be used to specify a precondition for when the method may not return. Suppose the following simple advice with specification. We disregard `signals` clauses for this discussion.

```
/*@
    requires true;
    ensures true;
    diverges false;
@*/

void around(): somePointcut() {
    proceed();
}
```

The specification states that the advice must return. Say that the pointcut matches calls to the method `someMethod` in the woven program.

```
void someMethod() {
    while (true)
        ;
}
```

The `proceed` call in the advice will refer to the method `someMethod`. This means that the implicit call to the advice at the matching join point will not terminate, contrary to what is stated in the specification of the advice. Of course, there is no way to know when verifying the advice that this situation will arise. Since it is impossible to demand that the advice should handle non-termination, we have to find another way of dealing with this problem. Two possible solutions are using partial correctness or changing the semantics of specifications. These solutions are presented below.

Translate to partial correctness

Partial correctness means, as mentioned in section 2.3, that the program is allowed to be non-terminating. If the program terminates and the precondition was fulfilled, then the postcondition must be true. However, the termination is not guaranteed.

When dealing with advice, a solution to the problem of non-returning calls to `proceed` could be to say that the translation of the specifications of advice should always be partial correctness proof obligations. This would allow the base program to diverge during calls to `proceed`. But, it would also allow the advice to diverge. An infinite loop could be placed in the advice without violating the specification. It would not be possible to talk about total correctness for the code in the advice. Especially when dealing with fault tolerance aspects this is not desirable.

An alternative is to use `diverges true` as default. But, according to the semantics of JML, this is equivalent to using partial correctness directly.

New interpretation of `diverges`

In this thesis the solution of giving new semantics to the `diverges` clause is favored over using partial correctness.

Here is a possible interpretation of specifications using `requires`, `ensures` and `diverges`. First, consider the case were the advice should not diverge.

```

/*@
    requires A;
    ensures B;
    diverges false;
@*/

<type> around(<args>): somePointcut(<args>) {
    <code>
}

```

The specification should be interpreted like this.

If A is satisfied when control flow transfers from a join point matching `somePointcut` to the *around* advice and *if all `proceed` statements in the advice terminate normally*, then the advice terminates normally and B is satisfied when the advice returns.

On the other hand, one might want to allow the advice to diverge for some precondition C.

```

/*@
    requires A;
    ensures B;
    diverges C;
@*/

```

The meaning of the specification is then.

If **A** is satisfied when control flow transfers from a join point matching `somePointcut` to the *around* advice, if all `proceed` statements in the advice terminate normally and the negation of **C** is satisfied, then the advice terminates normally and **B** is satisfied when the advice returns. If the predicate **C** is satisfied, then the execution of the code in the advice may not return to the join point.

Hence if the specification uses `diverges false` then we are using total correctness for the code in the advice, but still allow for the woven program to be non-terminating during the execution of the advice.

3.1.2 Exceptions

The `signals` clause in JML states postconditions for when methods return by throwing exceptions. This poses a problem when used in *around* advice, since it cannot be known when verifying what exceptions will be thrown during the execution of the base program. As an example of this, look at the following aspect and base program, where the same type of exception is thrown under different circumstances.

```
aspect SomeAspect {

    pointcut somePointcut(int[] arr) :
        call(int someMethod(int[])) && args(arr);

    /*@
       requires x.length >= 2;
       signals (ArithmeticException)
           x[0] == 0;
    @*/

    int around(int[] x) : somePointcut(x) {
        int result = proceed(x);
        result = result + 1/x[0];
        return result;
    }
}
```

In the base program the method `someMethod` looks like this.

```
int someMethod(int[] y) throws ArithmeticException{
    int result = 1/y[1];
    return result;
}
```

As has been specified, an exception will be thrown in the aspect if the first element of the array `x` is zero. But the same kind of exception will be thrown in the base program if the second element is zero. Assume an execution of the woven program where `x` has been defined like this before the call to `someMethod`.

```
x[0] = 1;
x[1] = 0;
```

In the call to `proceed` the program will try to divide by zero, hence an exception will be thrown. This exception is not caught by the advice which thereby returns exceptionally. When this happens, it will not be the case that the first element of `x` is zero as the specification states.

The conclusion is that it is not straightforward to use the `signals` clause as given by JML to specify advice. We obviously have no control over what exceptions might be thrown by the future base program. To our knowledge, the only way to use the same semantics of the `signals` clause as in JML is to have the advice catch every exception of the types used in the `signals` clauses from the call to `proceed`. Every `proceed` statement would have to be written like this.

```
try {
    proceed();
} catch (ExceptionType e) {
    <exceptionCode>
}
```

`signals_in_advice`

On the other hand, we might only be interested in what exceptions can be thrown in the advice and would want to disregard any exceptions that are thrown in the base program. For some purposes, it could make sense to specify an advice in this way. Especially since if the technique above were all exceptions from `proceed` are caught is used, then the exception handling from the base program would be overridden.

To accommodate this, we propose a second kind of `signals` clause. The clause would only be concerned with exceptions thrown in advice. Lets call the new clause `signals_in_advice`. The interpretation of

```
signals_in_advice (ExceptionType) S;
```

in advice specification would be

If the advice returns exceptionally by throwing an exception of type `ExceptionType` and the exception was thrown in the code of the advice not belonging to the base program, then `S` holds. If an exception of type `ExceptionType` is thrown during the call to `proceed` and it causes the advice to return, then it is not guaranteed that `S` holds.

With the introduction of `signals_in_advice` the `signals` clause retains its JML interpretation. So if `signals` is used, exceptions from the base program must be caught in the advice.

Motivation for `signals_in_advice`

Suppose we have this code segment in the base program.

```
try {
    doCalculations();
} catch (ArithmeticException e) {
    <exceptionHandling>
}
```

Couple this with an aspect advising calls to the method `doCalculations`.

```
aspect SomeAspect {
    pointcut Calculations() : call(void doCalculations());

    void around() throws ArithmeticException: Calculations() {
        proceed();
        <someStatements>
    }
}
```

Say that the advice will throw an `ArithmeticException` under some circumstances. Also, assume that we would like to specify this using a `signals` clause. Following the discussion above we would have to enclose the `proceed` call in a `try - catch` block to conform with the semantics of the `signals` clause in JML. But, there is code in place in the base program to handle exceptions. The exception handling block in the advice would take precedence over the base program's exception handling. The `signals_in_advice` clause allows to specify exception throwing in advice without this drawback.

3.1.3 Conclusion

To sum up the discussion on interpretation of specifications, here is how a piece of advice can be specified using the constructs mentioned above.

```
/*@
    requires A;
    ensures B;
    diverges C;
    signals_in_advice (ExceptionType1) S1;
    signals (ExceptionType2) S2;
@*/
```

This should be interpreted as follows.

Unless `A` holds nothing is guaranteed. If the advice terminates normally, then `B` holds. If the advice returns by throwing an exception of type `ExceptionType1` then `S1` holds, unless the exception originates from the base program. If the advice returns by throwing an exception of type `ExceptionType2` then the predicate `S2` will hold, regardless if it was thrown in the base program or in the advice. The advice may diverge if `C` holds. The base program is always allowed to diverge.

Compare this with the usual JML semantics (without the `signals_in_advice` clause, of course).

Unless `A` holds nothing is guaranteed. If the method returns by throwing an exception of type `ExceptionType2` then `E2` holds. If the method terminates normally, then `B` holds. The termination of the method is not guaranteed unless the negation of `C` holds.

Apart from the addition of `signals_in_advice`, the difference between the JML semantics and the one proposed in this thesis is that our semantics accommodates the non-termination of code not available when verifying.

3.1.4 Translation to Java Dynamic Logic

To give a more precise semantics of the specification it is here translated into Java Dynamic Logic (JavaDL). The translation gives a rough idea of how the informal semantics can be captured.

In the following we will use simplified pseudo code instead of real Dynamic Logic. That is, we won't use proof obligations exactly as they look or would look in the KeY Prover, since the goal is to illustrate the main principles. The pseudo code will have this form.

```
precondition -> {update} <program> postcondition
```

We only consider proofs which consists of a precondition, an update, a program and a postcondition. Rules are applied to symbolically execute the program. The proof is closed if the formula can be proved in the update (assignment to variables) given when the program has been removed by applying rules on it.

Compared to the KeY Prover we do not consider the sequent calculus turnstile \implies nor predicates to ensure initializations of objects and classes.

A specification of a usual Java method, `<method>`, as described in the previous section is translated into the following proof obligation. The predicates A, B, C and S2 refer to their JavaDL translations.

```
A -> [program] ((exc = null -> B) ^
  (exc != null -> (exc instanceof ExceptionType2 -> S2))) ^
A ^ ~C -> <program> true
```

[.] and <.> mean partial and total correctness respectively. `program` stands for

```
java.lang.Exception exc = null;
try {
  <method>
} catch (Exception e) {
  exc = e;
}
```

Except for the `signals` clause, the translation of a specification of an advice would be very similar. One way to achieve verification of the `signals` clause is to assume that an exception of the specified type is thrown in the `proceed` call during the symbolic execution. This is to ensure that the advice fulfills its specification even if the base program throws an exception of that type. Below, a possible translation of a specification of an advice into a proof obligation is shown.

```
(A -> [advice] ((exc = null -> B) ^
  (exc != null -> (exc instanceof ExceptionType1 -> S1) ^
  (exc instanceof ExceptionType2 -> S2)))) ^
(A ^ ~C -> <advice> true) ^
(A -> [adviceThrows] (exc = null -> B) ^
  (exc != null -> (exc instanceof ExceptionType1 -> S1) ^
  (exc instanceof ExceptionType2 -> S2)))
```

Where `advice` is the advice code in the same way as the method code was inserted above. `adviceThrows` stands for the advice code where each call to `proceed` is replaced with the throwing of an exception of type `ExceptionType2`.

When doing this, the methods to deal with the `proceed` call described in chapter 4 also has to be used. The code for both `advice` and `adviceThrows` is still encapsulated in `try - catch` blocks.

The major difference compared with the case for a method is that the advice code is included a third time. The proof obligation above is quite long, but serves to illustrate the issues arising when translating the specification of an advice into JavaDL.

3.2 Specification languages for AspectJ

The only specification language for AspectJ which we are aware of is Pipa [15]. Pipa is a BISL designed as a simple extension to JML. The most important constructs added with Pipa are a clause for specifying the `proceed` call (written `proceeds A`) and a keyword `then` to break up the specification into the parts before and after `proceed`. Clifton and Leavens [5] also deal with specification of AspectJ.

The aims of Pipa and [5] regarding specification language design are a bit different from ours. The goal of their approaches is to weave an AspectJ program and its specification into a Java program specified with JML. This is contrary to our need of handling both AspectJ and the specification directly in a verification system.

Pipa borrowed the `proceeds` clause from [5]. Despite this, there are some differences between the way Pipa and [5] specify *around* advice. The way specification is done in Pipa is shown below.

```
/*@
    requires A;
    proceeds B;
    then
    ensures C;
@*/
```

Clifton and Leavens use this specification style.

```
/*@
    requires A;
    ensures B;
    proceed(<arguments>);
    requires C;
    ensures D;
@*/
```

Neither of the approaches mention specification involving the `thisJoinPoint` construct. The problems of non-termination and exceptions described in section 3.1 are not addressed in neither Pipa nor [5] since they are not concerned with verification independent of base programs.

3.2.1 The `proceeds` clause

What is the exact semantics of the `proceeds` clause in Pipa? The paper on Pipa [15] says the following.

Pipa uses the proceeds *predicate* clause, taken from [5], to state that when control flow proceeds to the original method body (or to any additional advice if present), the *around* advice must make *predicate* hold.

According to this statement, the predicate belonging to `proceeds` is an obligation on the side of the advice. It is not a condition that the matching join point is supposed to fulfill. When symbolically executing advice, this means that the predicate must be fulfilled whenever a `proceed` statement is reached. It does *not* mean that the advice will proceed if the predicate holds when the advice begins executing. For example, the following advice with specification is correct.

```

/*@
    requires true;
    proceeds true;
    then
    ensures true;
@*/

void around() : somePointcut() {
    ;
}

```

An alternative view of the `proceeds` clause is to use it to specify under which precondition an *around* advice will proceed as an obligation of the join point. This would mean that if the predicate in the `proceeds` clause is satisfied when the advice begins execution, then the advice will execute the `proceed` statement at least once during the execution of the advice.

3.3 Specification of fault tolerance aspects

The majority of the aspects studied in this thesis are toy examples, designed only to illustrate difficulties with specification and verification of advice.

An example of a real fault tolerance aspect is given below. It is taken from [3]. The advice implements *time-redundant execution*, which is a technique to detect and mask transient faults by executing a method several times.

```

Object around() : TimeRedundantMethod() {
    RecoveryCache.establish();
    Object r1 = proceed();
    RecoveryCache.restore();
    Object r2 = proceed();
    if (!r1.equals(r2))
        throw new TransientFaultException();
    RecoveryCache.discard();
    return r1;
}

```

In the advice the join point is executed twice and the return values compared. A cache is used to save the state before each execution.

How would one specify this advice using the constructs available in JML? Normally, one could use `\result` to define what is computed. The problem is

that this advice is not concerned with *what* is computed, but with *how* it is computed.

Looking at this advice, it seems like we would need to use something to designate the return values of the `proceed` calls in the specification. Assume such a construct and call it `\proceedResult`. In fact, in this example there are two `proceed` statements, so two `\proceedResult`'s are needed. Now the advice can be specified as below.

```
requires true;
ensures \result.equals(\proceedResult_1);
signals_in_advice (TransientFaultException)
    !(\proceedResult_1).equals(\proceedResult_2);
```

In this example it is clear which `proceedResult` refer to which `proceed` statement. For more complicated situations problems will arise. For instance, if the `proceed` statement is within a loop it won't be straightforward to pair `proceed`'s with `proceedResult`'s. We have not investigated all details regarding how multiple `proceedResult`'s should be handled in general. Neither has the verification of this clause been considered.

Chapter 4

Formal verification

The previous chapter detailed how aspects can be specified. The next step is to design verification procedures to verify aspects according to their specifications. The method is formal verification by theorem proving via symbolic execution as in the KeY Prover.

We want to design a sound verification system. Soundness is in this case informally defined such that it should only be possible to verify an advice if it will fulfill its specification without dependence on the behavior of the base program it is woven together with.

The two most important problems in this respect are that the return value of `proceed` and the side-effects during its execution are unknown. It should only be possible to verify the advice if it is impossible for a base program to violate the specification via certain return values or side-effects.

This chapter will use the pseudo code introduced in subsection 3.1.4.

4.1 Handling the return value of `proceed`

Through the `proceed` call the action in the base program associated with the join point is executed. As with an ordinary method call, the join point may have a return value. When symbolically verifying an advice all possible return values from such a join point need to be taken into account. To see how this can be done, we can look at the way the KeY Prover handles input arguments to a method. When a method with arguments is verified, the symbolic execution must be performed for all possible values of the arguments. When constructing a proof obligation in the KeY Prover a universal quantification (“for all” quantification) is added to the Dynamic Logic formula. The quantification is over a logical variable, since program variables cannot be quantified. An “assignment” of this logical variable to the argument is placed in the update, which is the part of the formula representing the state. The same procedure is of course repeated for every argument. Informally, this means that arbitrary values are assigned to the arguments.

For verifying advice we can make use of the same principle. The rest of the proof is performed for any return value of `proceed` by inserting a quantification. In this section we assume that the base program terminates normally in the call to `proceed`.

4.1.1 Verification using quantification

In this section it is shown how to verify an advice which uses the `proceed` call such that the verification is independent of the value returned from a future base program. As example advice we use the fault tolerance advice from section 2.5 with the input array removed.

```
int around() : somePointcut() {
    int result;
    result = proceed();
    if (result < 0)
        result = 0;
    return result;
}
```

The specification is

```
/*@
    requires true;
    ensures \result >= 0;
@*/
```

The aim is to verify this advice according to the specification by making a quantification over the return value of `proceed`.

Initially the proof looks like this.

```
true -> {} <int result;
    result = proceed();
    if (result < 0)
        result = 0;
    return result;> \result >= 0
```

The next step is the `proceed` call. Here the idea of inserting a universal quantification is used. This gives the following situation.

```
all t_lv:int. true ->
    {result:=t_lv} <if (result < 0)
        result = 0;
    return result;> \result >= 0
```

This is provable and the rest of the proof can be handled by the rules already available in the KeY Prover.

4.1.2 Taclet

To verify advice in the KeY Prover rules for treating the `proceed` statement need to be introduced. The following simplified taclet (see section 2.3) assigns an arbitrary value to the location where the result of `proceed` is stored.

```
find (#allmodal{.. #loc=proceed(); ...}(post))
replacewith (all t_lv:typeof(#loc).
    ({#loc:=t_lv}#allmodal{.. ...}(post)))
```

The taclet states that it applies to situations where the result of `proceed` is stored in a variable. Use of `#allmodal` means that it applies both if we are proving total or partial correctness¹. The `replacewith` part says that the `proceed` statement can be removed while keeping the postcondition unmodified. In doing this, a quantification has to be inserted. This is done by first introducing a fresh logical variable `t_lv` of the type of the variable were the result of `proceed` is saved. Then an assignment of `t_lv` to this variable is added to the update.

The taclet only handles the situation where `proceed` is used in a direct assignment. However, this is no loss of generality since expressions using method calls are rewritten to save the results in variables during the symbolic execution in the KeY Prover.

There can be many `proceed` statements in the same program and each will get its own quantification.

Adding the taclet above would not be enough to be able to handle `proceed` in the KeY prover. It would have to recognize `proceed` as a keyword and not a method call, since there should not be any real method with that name.

The taclet deals with `proceed` with an empty arguments list. The same principle applies in the case of parameters. The prover would have to evaluate the parameters before discarding the `proceed` statement to take possible side-effects into account.

4.2 Side-effects of `proceed`

In this section the problem of how to take the possible side-effects of the `proceed` call into account when verifying is discussed.

4.2.1 Problem definition

Since we are verifying independently of any base program, it is crucial that the verification system will not verify an advice if it is possible that the specification of the advice is violated when the aspect containing it is woven together with a base program. In this subsection examples of when such violations can occur are presented. We will see that it is not enough to quantify over return values to achieve sound verification of advice.

When verifying a Java method in the KeY Prover and reaching a method call the code from the called method is inserted at the place of the call. Hence the proof handles all possible side-effects that the method call might have. The problem with independent aspects is that we do not have access to all the code which could potentially be reached during the execution of the advice. It is unknown when verifying what the calls to `proceed` will refer to. In this unavailable code, the base program might modify variables which are used in the advice. This poses a problem that has to be dealt with to be able to verify advice such that the verification is sound. Our goal is that it should not be possible to verify an advice with respect to a specification if it is possible that the woven program will violate the specification.

¹Those are the modalities considered in this thesis. In practice we might not want to apply the taclet for all possible modalities.

It is assumed that the base programs cannot access the aspects directly. Additionally, we assume that the aspects only can modify the parts of the base programs which are visible through the pointcuts.

First, we need to determine which variables in the advice are possible to modify from the base program. In the following paragraphs examples of how such modifications can occur are presented.

Arguments to the advice

Suppose we want to verify this advice.

```
void around(int[] x) : somePointcut(x) {
    x[0] = 0;
    proceed(x);
}
```

The specification of the advice is

```
/*@
    requires x.length > 0;
    ensures x[0] == 0;
@*/
```

Since there is no return value from `proceed` there is no need to insert a quantification when it is reached in the proof. Without taking possible side-effects into account, and since we have no base program, assume that the `proceed` statement is just removed from the proof. The update would then contain the assignment `x[0] := 0`, exactly what is stated in the postcondition. This would allow the proof to be closed.

However, this specification could potentially be violated when the aspect is woven with a base program. For instance, if we have a base program where `somePointcut(x)` matches a call to the method `someMethod` below.

```
void someMethod(int[] x) {
    x[0] = 1;
}
```

The program produced by the weaver will not obey the contract stated by the advice. The advice guarantees that when it has finished execution the first element of the array `x` in the argument is equal to zero. But, the call to `proceed` will transfer control flow to `someMethod` which will set the first element of `x` to one. Control flow then returns to the advice and immediately to the caller of `someMethod`. At this point the first element of the array sent as argument is not zero as promised by the advice.

The problem of the base program modifying arguments only arise when the arguments are of reference types. When the arguments are of primitive types, such as `int`, it is not possible for the base program to change the values of those variables in the advice. Additionally, it is not necessary that the advice parameter is sent back to the base program in the `proceed` call for the problem to appear. The base program can still change the value of the variable.

In the example the advice had access to variables from the base program through the argument to `proceed`. As will be shown below, it is not necessary that variables are explicitly sent to the advice for problems to occur.

Local variables in advice

It is also possible for the base program to modify a local variable in the advice, if the variable is of reference type. If there is a dependency between the local variable and the specification, then the woven program might violate the specification, as this example shows.

```

/*@
    requires true;
    ensures \result == 0;
@*/

int around(int[] x) : somePointcut(x) {
    int[] y = {0};
    proceed(y);
    return y[0];
}

```

This advice is similar to the previous one. The difference is that the advice does not send the argument to the join point to `proceed`, but instead sends a locally declared array. The array can then be modified in the base program. If this advice is woven to match calls to the method `someMethod` above the specification will be violated.

Fields of the aspect

The third problematic scenario concerns the fields of the aspect. In the example given here the base program does not need to know about the fields of the aspect. As was the case with arguments to the advice, the problem described is only valid for variables of reference types. Look at the following aspect.

```

abstract aspect SomeAspect {

    abstract pointcut somePointcut(int[] arr);
    abstract pointcut someOtherPointcut();
    static int[] x = new int[3];

    before(int[] y) : somePointcut(y) {
        x = y;
    }

    /*@
        requires x.length > 0;
        ensures x[0] == 1;
    @*/

    void around() : someOtherPointcut() {
        x[0] = 1;
        proceed();
    }
}

```

The aspect has a static integer array `x`. In the *before* advice it is assigned to an array sent from the base program. If the *before* advice executes before the *around* advice (note that they match different pointcuts), `x` will refer to the array from the base program. The base program can then alter the value of the array `x` in the call to proceed in the *around* advice, since `x` refers to an array available to it. Just looking at the *around* advice, it is impossible to know whether `x` can be altered in the call to `proceed` or not.

The specification of the advice will be violated if it is woven with this base program.

```
public class SomeClass {

    private static int[] z = {1,2,3};

    public static void getRef(int[] y) {
        ;
    }

    public static void useRef(){
        z[0] = 2;
    }

    public static void main(String[] args) {
        getRef(z);
        useRef();
    }
}
```

For this example it is assumed that `somePointcut(int [] arr)` matches calls to `getRef(int [] y)` and that `someOtherPointcut()` matches calls to `useRef()`.

The call to `getRef` will make `z` in the class `SomeClass` refer to the array `x` in the aspect. The call to `useRef` will then set the value of the first element of this array to two. But, the *around* advice, which will execute around the call to `useRef`, states in its specification that the first element of `x` will be set to one when it returns. This will not be the case.

Objects from thisJoinPoint

Also, use of `thisJoinPoint` might cause problems with side-effects. The `thisJoinPoint` variable provides reflective information about the join point where the advice is executing. This can be used to access, among other things, the arguments to the join point. Calls to `thisJoinPoint.getArgs()` return an array with the arguments to the join point. These can be arguments which are not visible in the advice header. Since we are interested in independent aspects, the pointcuts we are dealing with are abstract (an aspect is tied to a base program when the pointcut is instantiated). Thus the arguments are not visible in the pointcut definition either.

When the advice is written, there is no base program to refer to. Hence the advice cannot know what types of variables will be accessible through `thisJoinPoint`. This means that the code which handles these arguments need to be very general.

`thisJoinPoint` is discussed in section 4.3. Since it is unclear how to handle it in general, use of the particular method `getArgs` is not considered further. Additionally, the return value from `getArgs` will have to be assigned to a variable to be used in advice. This variable will fall under one of the categories described above.

Conclusion

We have to design our verification for advice such that advice which suffer from the problems described above will not be verifiable. It is use of the following kinds of variables which pose problems.

- Advice parameters of reference types
- Local variables in advice of reference types passed to `proceed`
- Fields of the aspect

A way in which sound verification of advice can be achieved with these problems in mind is described in the next subsection.

4.2.2 Introduce universal quantifications

A method to handle side-effects would be to insert quantifications, similar to the way the return value of `proceed` is handled in section 4.1.2. The idea is to add universal quantifications over the kinds of variables which can cause the problems described in subsection 4.2.1. Whenever a `proceed` call is reached in the symbolic execution, the rest of the proof must be performed for all possible values of these kinds of variables. Informally, this means that every assignment the base program could do in the woven program is considered in the proof.

To illustrate how this would be done, assume that we want to verify the advice below.

```

/*@
  requires x.length > 0;
  ensures x[0] == 0;
@*/

void around(int[] x) : somePointcut(x) {
  proceed(x);
  x[0] = 0;
}

```

Our goal is to be able to verify this advice *only* if it will fulfill its specification no matter what side-effects take place during the call to `proceed` in the woven program. Looking at the advice and its specification, it should be possible to verify it.

For the proof, it is assumed that the call to `proceed` returns normally. See section 3.1 for a discussion on the interpretation of specification clauses.

Initially, the proof looks like this.

```

all t_lv:ArrayOfInt. x.length > 0 ->
  {x:=t_lv} <proceed(x);
  x[0] == 0;> x[0] == 0

```

The next step is the `proceed` call and the quantification of the argument is made. There is of course no need to quantify over return values since the advised join point is of type `void`.

```
all t_lv:ArrayOfInt. all u_lv:ArrayOfInt.
  x.length > 0 ->
    {x:=t_lv, x:=u_lv}
    <x[0] = 0;> x[0] == 0
```

In fact, the quantification was redundant, the array `x` had already been assigned an arbitrary value since it is an argument.

In the KeY Prover the situation where the index is out of bounds has to be handled in an assignment to an array. For this example, we simplify matters and just move `x[0] := 0` to the update.

```
all t_lv:ArrayOfInt. all u_lv:ArrayOfInt.
  x.length > 0 ->
    {x:=t_lv, x:=u_lv, x[0]:=0}
    <> x[0] == 0
```

This should be provable. The last addition to the update sets the first element of the array `x` to zero, exactly what is stated in the postcondition.

To get an advice where the quantification is necessary to make the advice implementation non-verifiable with respect to the specification, we can have the two statements in the advice above switch order.

```
void around(int[] x) : somePointcut(x) {
  x[0] = 0;
  proceed(x);
}
```

Again, the KeY Prover again has to deal with a possible index out of bounds exception in the assignment to `x[0]`. We take a shortcut and move the assignment to the update.

```
all t_lv:ArrayOfInt. x.length > 0 ->
  {x:=t_lv, x[0]:=0} <proceed(x);>
  x[0] == 0
```

The same quantification as above is inserted.

```
all t_lv:ArrayOfInt. all u_lv:ArrayOfInt.
  x.length > 0 ->
    {x:=t_lv, x[0]:=0, x:=u_lv} <> x[0] == 0
```

At this point, it has to be proved that the first element of `x` is zero, but according to the state, `x` has an arbitrary value. The proof cannot be closed.

4.3 The `thisJoinPoint` construct

The `thisJoinPoint` variable can be used within advice and it is desirable to have means to handle it formally. In [3] there is an example of a fault tolerance aspect where `thisJoinPoint` is used. The aspect implements an *incremental*

recovery cache. There are no specifications in the paper, but `thisJoinPoint` would likely be useful in the specification of the recovery cache.

The recovery cache aspect is quite complicated. To illustrate the use of `thisJoinPoint` both in specification and inside the body of advice the simpler aspect shown below is used. The aspect saves the signature of the join point in a string. There are several other methods similar to `getSignature()` in the `thisJoinPoint` interface.

```

abstract aspect ThisJoinPoint {
    abstract pointcut Calculations();
    String tjp;

    /*@
        requires true;
        ensures tjp == thisJoinPoint.getSignature().toString();
    @*/

    void before():Calculations {
        tjp = thisJoinPoint.getSignature().toString();
    }
}

```

It is possible in JML to use pure, or side-effect free, method calls in specifications. The nature of the methods available to a `thisJoinPoint` object implies that they are pure.

How do we treat `thisJoinPoint` in a verification system such as the KeY Prover? When reaching a method call in KeY the code belonging to the method is inserted at the place of the call. But, when verifying aspects we don't have access to the signature of the join point. Even if the code of the methods in `thisJoinPoint` were inserted, at some point a value not available until runtime would be needed.

4.3.1 Verify for all return values

One way of handling `thisJoinPoint` is to insert a universal quantification over the variable to which its return value is assigned. This is in the same vein as the return value of `proceed()` could be handled as described in section 4.1.2. However, this technique would not be a very precise model of how `thisJoinPoint` works. It would not be possible to verify the advice above, since the method call in the ensures clause also would add a quantification.

On the other hand, assuming that `thisJoinPoint` does not appear directly in specifications this could be a suitable method for treating the construct in a verification system.

In conclusion, we do not have satisfactory methods to deal with `thisJoinPoint`.

Chapter 5

Result and discussion

5.1 Result

In this report the problem of specifying and verifying aspects which are independent of any base program was analyzed. Methods to handle the complications introduced when verifying a piece of advice which can be combined with different base programs was presented. This section briefly sums up the contributions of this thesis.

5.1.1 Specification

The major problem regarding specification is that it is unknown whether the call to `proceed` will terminate normally, terminate exceptionally or not return at all.

To handle exception throwing we suggest to introduce a new specification clause, `signals_in_advice`. This clause can be used to specify a postcondition for when the code in the advice itself returns exceptionally. Its interpretation is that its predicate is only guaranteed to hold if the exception occurred in the advice code. That is, it did not occur during the call to `proceed`. The `signals` clause keeps its JML interpretation implying that the exceptions of the types used in the specification need to be caught from the `proceed` call.

The interpretation of the specifications also needs to accommodate non-returning calls to `proceed`. This leads to modifying the semantics of the `diverges` clause in JML when it is used with advice. In its new interpretation it only applies to the code in the advice. The code in the advice may only diverge if the corresponding precondition is fulfilled, but the call to `proceed` is always allowed to diverge. This can be seen as analogous to the `signals_in_advice` clause.

5.1.2 Verification

To achieve sound verification of advice two problems and possible solutions to them are considered.

Since the return value of the future join point is not available when verifying, the proof has to be performed for all possible values that could be returned. In the KeY Prover, this can be done inserting a universal quantification over a fresh logical variable. An assignment of this new variable to the variable where

the return value is saved is placed in the update. This means that the advice can only be verified if it fulfills its specification for all return values.

The variables which can be modified in the call to `proceed` are handled according to the same scheme. When a `proceed` statement is reached in the proof, the rest of the proof must be performed for all possible values of the following kinds of variables.

- Advice parameters of reference types
- Local variables in advice of reference types passed to `proceed`
- Fields of the aspect

This can be done through universal quantification. In this manner we hope to provide sound verification of advice.

5.2 Discussion

This thesis does not design a full verification system for aspects. The areas where solutions are provided are the interpretation of specification clauses concerning exception throwing and diverging, as well as how to handle return values and side-effects.

5.2.1 Loose ends

Our proposed solutions do leave some loose ends which will be discussed here.

Verification of the signals clauses

Although a possible translation of specification to a proof obligation was presented in subsection 3.1.4, it is not clear how to verify exception throwing in the context of advice. This area still needs more work. In particular, it is the use of the ordinary `signals` clause which cause problems. If it is ignored the proof obligation for an advice becomes very similar to the case for a method.

Verification of the proceeds clause

The `proceeds` clause was discussed in section 3.2.1. In this thesis a different interpretation than the one given in Pipa is presented. If the original semantics is used, it would mean that the predicate in the clause would have to be proved every time a `proceed` statement is reached during symbolic execution. Likely, this leads to a branch of the proof.

An alternative semantics is to use the `proceeds` clause as a precondition (similar to `diverges`). If the predicate holds when the advice begins execution, then the advice will proceed to the base program at least once. This appears somewhat less straightforward to prove.

Other missing pieces

We briefly introduced and argued for a new clause called `\proceedResult` in section 3.3. It would definitely require more investigation to determine whether it is useful and possible to implement.

Also, we lack a solution for how to handle `thisJoinPoint` in verification of advice. It seems likely that useful advice can be written without using it, but it is of course desirable to be able to deal with it formally. It is, however, unclear if it is even possible to do so.

5.2.2 Limitations

With the approach taken in this thesis sharing between the aspects and the base programs becomes very limited. In fact, only what is exposed by pointcuts can be shared. It would be difficult to use any globally accessible objects, such as the Java standard library. It could be possible if side-effects are taken into account by quantification of global variables as in section 4.2, but this would hardly be useful or desirable.

When designing the verification methods we have strived not to assume anything of what the base program will do. The semantics of the specifications are adjusted to this. We also believe that the verification is sound with respect to return values and side-effects. But, our methods have not been tested or implemented at all. It is hard to say at this stage what limitations will be revealed when and if verification of advice as presented here is implemented. It is also hard to predict what difficulties will appear should a further step be taken and our methods applied to more realistic systems.

5.2.3 Is it useful?

It remains to be seen whether it is feasible in practice and useful to verify aspects independently. A most important question is if the solutions presented here are a step in the right direction. Perhaps the only viable way to verify aspects is to verify together with a base program. In such a scenario, verification as we propose in this thesis is not relevant at all. The source of our problems is that we do not know what the `proceed` call refers to. This would of course not be unknown with a base program available.

Similarly, it might turn out that useful verification of advice cannot be achieved unless some things are assumed about the future base program. This possibility has not been investigated in this thesis.

On the other hand, to verify fault tolerance nothing can be assumed about the behavior of the program to which the fault tolerance mechanisms are to be applied.

It might also be the case that many aspects do not lend themselves to be specified in the way methods in JML are specified. This was discussed in section 3.3.

5.2.4 Future work

Apart from tying up the loose ends mentioned above, with verification of `signals` clauses being the most important, it would be interesting to investigate possible implementations of our verification of advice. The KeY Prover would be the natural starting point, but the methods given here might be sufficiently general to be applied in other verification systems.

It is also vital to investigate if the solutions provided here can be applied to more realistic aspects, other than the toy examples discussed in this thesis.

Perhaps the most interesting future work is to investigate an alternative approach to verify aspects. One could either verify an aspect given a (partial) specification of a base program or verify an aspect and produce a contract which the base program would have to obey. It is mostly the problem of side-effects that makes this a more desirable approach. The `\assignable` clause in JML, which states which fields may be assigned to, would likely be useful. A possible scenario could be to verify an advice given an `\assignable` clause.

5.2.5 Conclusion

It seems realistic to specify and verify independent aspects. The methods detailed in this thesis provide solutions to some of the more fundamental questions. Despite this, several areas need more investigation. It is necessary to determine whether it is indeed useful to verify independent aspects with the methods presented here.

Bibliography

- [1] AHRENDT, W., BAAR, T., BECKERT, B., BUBEL, R., GIESE, M., HÄHNLE, R., MENZEL, W., MOSTOWSKI, W., ROTH, A., SCHLAGER, S., AND SCHMITT, P. H. The KeY tool. *Software and System Modeling 4* (2005), 32–54.
- [2] ALDRICH, J. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In *FOAL: Foundations Of Aspect-Oriented Languages* (Mar. 2004), C. Clifton, R. Lämmel, and G. T. Leavens, Eds., pp. 7–18.
- [3] ALEXANDERSSON, R., ÖHMAN, P., AND IVARSSON, M. Aspect oriented software implemented node level fault tolerance. In *Ninth IASTED International Conference on Software Engineering and Applications (SEA 2005)* (Phoenix, AZ, USA, Nov. 2005).
- [4] BECKERT, B. A dynamic logic for the formal verification of Java Card programs. In *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France* (2001), I. Attali and T. Jensen, Eds., LNCS 2041, Springer, pp. 6–24.
- [5] CLIFTON, C., AND LEAVENS, G. T. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002* (2002), G. T. Leavens and R. Cytron, Eds., no. 02-06 in Technical Report, Department of Computer Science, Iowa State University, pp. 33–44.
- [6] DENARO, G., AND MONGA, M. An experience on verification of aspect properties. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution* (New York, NY, USA, 2001), ACM Press, pp. 186–189.
- [7] GIESE, M. Taclets and the KeY prover. In *Proc. User Interfaces for Theorem Provers Workshop, UITP 2003* (2004), D. Aspinall and C. Lüth, Eds., vol. 103 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 67–79.
- [8] HAREL, D., TIURYN, J., AND KOZEN, D. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.
- [9] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP*

- 2001, *LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.
- [10] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. Tech. Rep. SPL97-008 P9710042, Xerox Palo Alto Research Center, Feb. 1997.
- [11] LARSSON, D., AND ALEXANDERSSON, R. Formal verification of fault tolerance aspects. In *Supplementary Proceedings, International Symposium on Software Reliability Engineering (ISSRE) Conference 2005, Chicago, USA* (2005), pp. 279–280. Fast Abstract.
- [12] LEAVENS, G., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., AND MÜLLER, P. JML reference manual (draft), July 2005.
- [13] LEAVENS, G. T., BAKER, A. L., AND RUBY, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publishers, 1999, pp. 175–188.
- [14] SIHMAN, M., AND KATZ, S. Model Checking Applications of Aspects and Superimpositions. In *FOAL: Foundations of Aspect-Oriented Languages* (Mar. 2003), G. T. Leavens and C. Clifton, Eds., pp. 51–60.
- [15] ZHAO, J., AND RINARD, M. Pipa: A behavioral interface specification language for AspectJ. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003), LNCS 2621* (Apr. 2003), Springer-Verlag, pp. 150–165.