Universiteit Antwerpen

Static Contract Checking in Aspect Oriented Programming Languages

Luis Mayorga

Principal Adviser: Dirk Janssens
Assistant Adviser: Tim Molderez

Dissertation Submitted in May 2014 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Contents

List of Figures

Acknowledgements

This thesis is submitted during the academic year 2013-2014 at the University of Antwerp as part of an Erasmus Lifelong Learning Programme exchange with the Technical University of Madrid. For this reason, I have to thank all the people that made it possible for me to have such a good experience.

I would like to thank Tim for his great help during the elaboration of this thesis. Without his time and guidance it would not have been possible. I also have to thank Javier for helping me during this year with all that he could.

Last but not least, I owe a huge debt of gratitude to my parents and sister for making it possible for me to study what I liked and give me all the facilities that they could as well as supporting me in all my projects.

Abstract

Aspect-oriented programming is an extension to object-oriented programming that makes it possible to separately define crosscutting functionality that should be executed whenever particular methods are called. However, if it does not preserve the behaviour of the method, it can produce unexpected behaviour. This thesis takes the work on an approach to modular reasoning for aspects by Molderez and Janssens, and introduces a technique to statically check whether their approach is satisfied. It is shown how to check restrictiveness of the contracts present in the software in order to know whether the desired behaviour is kept, as well as the process that must be applied to Java contracts. A proof of concept implementation of this technique is presented as well.

CHAPTER 1

Introduction

Aspect-oriented programming (AOP) is often realized as an extension to the object-oriented programming (OOP) paradigm that makes it possible to modularize crosscutting concerns. We understand crosscutting concerns as parts of a program that affect many other parts of it and that cannot be detached from them.

Modules called aspects allow such crosscutting concerns to be defined as a separate module. Those aspects can then be developed independently from the parts where they are required, without tangling code regarding different domains and weaving it together to create an advised object.

Being able to separate these crosscutting concerns from other concerns makes it possible to avoid scattering the code relative to a specific feature along all the software components that make use of it, as happens in OOP. This can improve the separation of concerns and can reduce software complexity in certain parts. Common simple uses of aspects could be logging or authentication but they can become much more complex [11].

Although this is a great advantage in many of the cases, it also has some drawbacks. The behaviour of a method call no longer depends on just the method's definition, but also on all the aspects that advise it. Therefore, it is necessary for the developer to for the developer to be aware of all aspects in order in order to check that advised methods will still behave according to what is expected of them, hindering modular reasoning. Modular reasoning can be understood as the ability to reason about a module, by looking only at that module itself (and the modules it explicitly refers to).

Design by contract makes it possible to describe the behaviour of a software component in terms of preconditions, postconditions and invariants [16]. This also applies to Aspect-oriented software development (AOSD), which allows us to formalize the behaviour of the aspects and the advised methods. Analogous to the concepts of Behavioural Subtyping and the Liskov Substitution Principle [15] in OOP, a similar set of rules can be defined in AOP.

Although the notion of an Advice Substitution Principle (ASP) has been around for some time [22], it was formalized by Molderez and Janssens [18] and stands in their work as one of the main elements that are necessary to maintain modular reasoning in AOP. The ASP specifies all the conditions that must be met such that aspects do not cause any surprising behaviour in relation to the advised methods. In cases where the rules of the ASP cannot be satisfied, the use of a new specification is proposed, `@advisedBy`, as a way of restoring modular reasoning. With this this approach to modular reasoning, they define a minimal aspect-oriented language, ContractAJ, and a runtime contract enforcement algorithm.

This thesis uses the mentioned work as the starting point and tries to inquire on the possibility of statically checking the same principles. Static checking allows us to check all the possible executions of a program without the need of executing it, which provides a great advantage. With this purpose in mind, this thesis introduces a technique to compare the strength of contracts and its use to statically check modular reasoning in AOP. We also provide a proof-of-concept prototype plugin that implements this technique for AspectJ.

1.1 Scope of the research

The objective of this thesis is to find out whether it is possible to statically check modular reasoning in an AOP context. In order to check modular reasoning, rules such as the ASP need to be checked, which involves checking whether the contracts of one module are stronger or weaker than those of another module. For this reason, we will study how contracts can be compared in this way and try to provide a prototype able to do such a comparison for AspectJ. The next issues will be addressed in order to answer the previous question.

First, we give an overview of the Advice Substitution Principle and how modular reasoning in AOP is faced in Molderez and Janssens' work [18]. Then,

a study of how contracts can be analyzed to achieve the same soundness in a static manner as well as the comparison procedure, converting Java contracts to SMT-LIB formulae. Finally, the implementation of a prototype able to prove that the approach is possible.

1.2 Structure of the document

The remainder of this document is structured as follows:

The second chapter discusses the background necessary to understand the aim and the outcome of this thesis along with the basics of design by contract and behavioural subtypes in OOP and in AOP. It also covers the conditions required to maintain modular reasoning in an AOP environment.

The third chapter addresses the process to check these conditions statically, including how the strength of two contracts can be compared statically and how this allows to check whether the ASP is met.

The fourth chapter describes the tool's development, the design decisions that were made and how to use the tool.

The fifth chapter contains an overview of similar works and how their how their approach relates to ours.

Finally, the sixth chapter presents the conclusion and future work.

CHAPTER 2

Background

This chapter addresses the basic concepts used throughout this thesis. OOP and AOP are briefly explained, relating both to Design by Contract and behavioural subtyping. This is followed by a review of how to achieve modular reasoning in AOP.

2.1 Design by contract

Design by Contract provides a systematic and modular approach to specify and implement software elements and their specifications in a software system [16]. It suggests associating a specification to each software element within a system. The system is viewed as a set of components whose interactions are based on precisely defined specifications of their mutual obligations or contracts. In such a system, the specification of those contracts governs the interaction interactions among software components.

Design by Contract provides numerous benefits. It helps to improve the quality of the software, improving its reliability by checking whether the implementation satisfies its specifications. It also makes it possible to reuse components whose correctness is already proven and trusted. From a quality assurance perspective, it makes it easier to test and find errors, because there exists a definition of what is expected of the software module. Finally, it is a good way of documenting software, as it defines the interface of a module, like a short form stripped of all implementation information [17].

This specification between client and supplier modules are defined by means of preconditions, postconditions and invariants. The contracts available are:

@requires Defines a precondition. That is, a condition that must be met prior to the execution of the method.

@ensures The postcondition must be satisfied after the execution of the method.

@invariant This conditions should always be met within the class.

Contracts can be used in many programming paradigms. In some languages they are available as an part of the language itself (e.g. Eiffel), while in others support can be added through frameworks or external tools. In Java, the language used for out prototype, contracts can be embedded into the code as annotations.

2.2 Object Oriented Programming

2.2.1 Contracts in OOP

As explained in Section 2.1, in many programming paradigms it is possible to define the behaviour of a software module through the use of some specification clauses and this also applies to object-oriented software. In this thesis, Java will be used due to the facilities that provides, the vast amount of libraries and frameworks available for it and its flexibility to annotate source code.

It is possible to define `@requires` and `@ensures` clauses for preconditions and postconditions respectively, as well as `@invariant` for class invariants. These contracts are included in the source code as Java annotations. Although `@invariants`, as stated, belong to a class, they apply during execution of all the methods contained in that class. Taking that into account, for the sake of simplicity, they will be considered for each contract, as pre-and postconditions are shared by all the members of a class. A simple example of an annotated class is shown in Fig. 2.1.

In line number 5 a public method `move`, annotated with preconditions and postconditions, can be seen.

```
1 public class Train{
2
3         @requires("entity.isAvailable()")
4         @ensures("entity.isBusy() && entity.isAnnounced()")
5         public void move(RailEntity entity){...}
6 }
```

Figure 2.1: Annotated contracts in a class.

2.2.2 Behavioural subtyping in OOP

In OO programming, a subtype can be any class that extends the functionality or inherits the behaviour of another class. As Liskov and Wing [15] stated, the extending class can be used according to its apparent (inherited) type with the expectation that, if the program works correctly with a type, it should also do so with its subtype. This possibility of being able to seamlessly swap two types of the same hierarchy is known as Liskov Substitution Principle (LSP).

However, there is a problem that might arise. A class can syntactically be a subtype of another class, but this does not ensure that it also is a behavioural subtype. This means that the new class, although defined as a subtype could behave in a different way than the superclass, causing unexpected behaviour. Classes that do not comply with the LSP produce unexpected behaviour, given that the developer should only rely on the specifications of the receiver's static type whenever a method is called.

Since specification of behavioural interfaces can be achieved in OOP using contracts, as explained in Section 2.2.1, it is possible to analyse behavioural subtyping from a Design by Contract perspective [6; 12].

The definition of the behavioural interface allows to strictly compare methods and uncover misleading inheritances. To explain how this is possible, it is necessary to introduce the concept of *refinement* as a binary relation on method specifications. Below is the definition given in [12]:

Let $T' \triangleright spec'$ and $T \triangleright spec$ be specifications of an instance method m, such that T' is a subtype of T. Then $spec'$ refines $spec$ with respect to T', written $spec \sqsupseteq^{T'} spec$, if and only if for all calls of m where the receiver's dynamic type is a subtype of T', every correct implementation of $spec'$ satisfies spec.

We can then consider that a class is a behavioural subtype of another when the method specifications of the subclass refine the ones in the superclass and when the invariant of the subclass implies the invariant of the superclass [12].

```
1 public class CargoTrain extends Train{
2
3         @requires("platform.isAvailable() && platform.isCargo()")
4         @ensures("platform.isBusy()")
5         public void move(PlatformEntity platform){}
6 }
```

Figure 2.2: Inheriting class.

```
1         RailEntity pe = new PlatformEntity();
2         pe.setAvailable(true);
3         pe.setCargo(false);
4         Train t = new CargoTrain();
5         t.move(pe);
```

Figure 2.3: Call causing violation of method contracts.

This can be summed up in the following rules. Making a subclass a Strong Behavioural Subtype (SBS) of another class means that within it:

- Method preconditions must not be strengthened.
- Method postconditions cannot be weakened.
- Class invariants should be preserved.

An example of how neglecting classes to be behavioural subtypes could cause problems is the following. Fig. 2.2 shows the `CargoTrain` class, which extends the example given in Fig. 2.1. This class overrides the `move` method but imposes different constraints on it. In this inheritance hierarchy, the `@requires` contract adds another variable, so the clause is stronger than the one being overridden. This means that it is more difficult for the precondition of the subclass to hold. The same happens with `@ensures`, but in this case, the condition is weakened and for that reason, the state ensured after the execution of the method is less strict.

An example of a method call that would make the contracts of the superclass hold, but not the ones defined in the subclass is shown in Fig. 2.3. The statement on line number 2 makes the precondition of the method in `Train` true. However, the one in line 3 makes the precondition of the overriding method not hold. For this reason any part of the code where `Train` is the static type expected and this method is called, would not work as it should if it receives a `CargoTrain` object.

2.3 Aspect Oriented Programming

Although in object-oriented software development requirements can belong to different domains and can evolve independently, the implementation tangles them together, weakening the conceptual separation between concerns that existed at design time [11].

At the same time, a single cross-cutting concern that affects several modules will have its code scattered along all the components where its functionality is required. Changes regarding these features are propagated to all the spots where its implementation is fragmented and thus, cohesion is reduced.

Aspect Oriented Programming (AOP) intends to solve this by making it possible to separate those cross-cutting concerns from the concerns they interact with. AOP is just a methodology and it means that, in order to be of any use, it must be implemented. Each existing implementation of AOP consists of two main parts. One is the specification that describes the language constructs and the syntax expressing the implementation of the core and cross-cutting features. The second one is the implementation itself, that verifies that the code adheres to the specification given and makes it executable. The implementation used in this thesis is AspectJ [7].

Next to using AOP, it is possible to use other methods to get the similar benefits. Using some frameworks or certain design patterns also makes it possible to modularize concerns with some limitations. However, the complexity of the software is dramatically increased. For this reason, such solutions can be cumbersome to use for developers, but are still valid mechanisms for the internal implementation of aspect oriented reasoning.

Using AOP has several benefits. One of them is that it greatly helps to simplify the design and allows for cleaner implementation due to the improved separation of concerns. Another one is that it enables for better code reuse, as responsibilities are better distributed and the cohesion is higher.

2.3.1 Main AOP Concepts

Some basic elements of an AOP system are the following:

Join point Each one of the points that the program exposes at runtime. These points can include method calls, creation of objects, etc.

Join point shadow The location in the source code of a join point.

Pointcut Construction used to quantify. A pointcut selects a set of join points. It can be seen in line 3 of Fig. 2.4.

Advice Defines the functionality added in the join points selected by an associated pointcut. An example is shown in line 6 of Fig. 2.4.

Aspect The logical unit that encapsulates the cross-cutting concerns. A declaration of an aspect is shown in Fig. 2.4.

Inter-type declarations Declaration of class members, where the declaration statement itself is located in an aspect.

```
1 public aspect SafetyMonitor {
2
3         pointcut moves(RailEntity entity) :
4                 execution(void *Train.move(*Entity)) && args(
                      entity);
5
6         void around(RailEntity entity) : moves(entity){...}
7 }
```

Figure 2.4: Basic aspect and its elements.

Aspects and advised classes are woven together using the rules defined by the aspects. Depending on the implementation, interlacing the core functionality and the cross-cutting concerns can happen in code through the compiler, or in the bytecode.

2.3.2 Contracts in AOP

As AOP is typically an extension of OOP, it is possible as well to attach contracts to the code, which specify the behaviour of each module. Aside from attaching contracts to methods and classes, as described in Section 2.2.1, they can be attached as well to aspects and advice.

In addition, there are other elements that can take contract annotations. These are the inter-type declaration of members, that for this purpose act as regular methods and the advice declared in aspects. Both of them accept the same annotations as in-class declared methods.

An annotated aspect that belongs to the same system as Fig. 2.1 can be seen in Fig. 2.5. This aspect is in charge of the safety in the system and whenever a move is attempted it checks whether it is safe to perform it.

```
1 public aspect SafetyMonitor {
2
3         @requires("entity.available() && entity.isMonitored()")
4         @ensures("(entity.hasTrainBounded()?true:proc)")
5         void around(RailEntity entity) :
6                 call(void *Train.move(*Entity)) && args(entity)
                      {...}
7
8 }
```

Figure 2.5: Contracts in Aspect Oriented Programming.

2.3.3 Substitution principle in AOP

As in OOP, there exists a notion of a substitution principle in Aspect Oriented Programming which is quite similar to the one described in Section 2.2.2.

Advised methods contain the core functionality or business logic of a program and can be extended by aspects that implement cross-cutting functionality. Calls to advised methods are intercepted and the corresponding advice are executed. However, these calls are implicit and other modules using the main functionality may not be aware of their existence. If the executed advice does not behave as a behavioural subtype of the advised method in each one of the pointcuts, it may produce an unexpected result.

The purpose of the around advice shown in Fig. 2.5, is to prevent the move from happening if it is not safe. This is achieved by skipping the call to `proceed` within its execution if the conditions for it are met. Its consequences can be seen in its `@ensures` clause, which indicates that no specific state can be expected, specifying a weaker postcondition than the wrapped method. Something similar happens with the `@requires` contract, which has an added constraint on the advice, `entity.isMonitored()`, due to the need of the entity to be monitored in order to perform the safety checking tasks.

The consequence of these changes in the restrictiveness of the contracts is that the advice is no longer a behavioural subtype of the advised method and any call to the `move` method would trigger the execution of the advice with unexpected results.

Fig. 2.6 shows some code that corresponds to the previously described scenario. Line 3 of that code has no meaning for the method. However, it causes the precondition of the advice not to hold and yet, it is executed. This reasoning also applies when the advised element is another advice instead of a

```
1                RailEntity pe = new PlatformEntity();
2                pe.setAvailable(true);
3                pe.setMonitored(false);
4                Train t = new Train();
5                t.move(pe);
```

Figure 2.6: Implicit call of an advice and violation of the method's contracts.

method.

2.4 Modular Reasoning in AOP

Molderez and Janssens present in their work how to preserve modular reasoning in AOP from a design by contract perspective [18] and provide a runtime contract enforcement algorithm. There are some concepts in it that are useful for this thesis.

They define ContractAJ, a language used to study modular reasoning in the context of aspects. One of the reasons for defining it is that, since it is a minimal language, it is better suited to study modular reasoning on a more formal level. Another is that this language allows to ignore the design decisions driven by the necessity of implementing AspectJ, something that makes it more abstract and less rigid. For this reason, the arguments presented use this language as a basis.

The conditions describing strong behavioural subtyping in OOP, seen in Section 2.2.2, also apply in similar fashion to advice. These conditions are formalized in the Advice Substitution Principle (ASP) for all advice kinds. The conditions for each kind are listed in Molderez and Janssens' work as follows.

ASP for around advice

Consider an around advice in type t that is applied to join point u.x, representing a method call or an advice execution. If x is a method, u is the static type of the receiver. If x is an advice, u is the class containing x. The around advice satisfies the ASP if and only if, for all objects of type t:

- *The advice's precondition must be equal to or weaker than the precondition of u.x.*

- *The advice's postcondition must be equal to or stronger than the postcondition of u.x, if the precondition of u.x held in the pre-state.*
- *The advice's invariant must preserve the invariant of u.*

ASP for before advice

- *The advice's precondition must be equal to or weaker than the precondition of u.x.*
- *If the precondition of u.x held before executing the advice, it should still hold after the advice (at the implicit proceed call). This implies the advice's postcondition may not invalidate u.x's precondition.*
- *The advice's invariant must preserve the invariant of u.*

ASP for after advice

- *The advice's precondition must be equal to or weaker than the postcondition of u.x.*
- *If the postcondition of u.x held before executing the advice, it should still hold after the advice (at the implicit proceed call). This implies the advice's postcondition may not invalidate u.x's postcondition.*
- *The advice's invariant must preserve the invariant of u.*

2.4.1 AdvisedBy clause and proc keyword.

One important feature of ContractAJ that complements the ASP is the presence of the `@advisedBy` clause. There are some advice like the `SafetyMonitor` example shown in Fig. 2.5 that, due to their nature, cannot be rewritten in an ASP compliant form. One way to solve this would be to disregard AOP in favour of plain method calls. However, putting aside AOP and its benefits without looking for a solution just for this reason might be excessive.

In those cases, `@advisedBy` can restore modular reasoning. This annotation is placed on a method and lists its non-ASP advice. Thus, other advice can be aware of the presence of these advice and expect contracts to be broken by them.

Given the example shown in Fig. 2.7, the aspect `SafetyMonitor` displayed, due to its postcondition, could cause the `move` method not to be executed. This would be marked on it using the `@advisedBy` annotation.

Another characteristic introduced in their work is the use of the keyword `proc` for specification inheritance. This keyword represents the same contract

```
1 public class Train{
2
3         @requires("entity.isAvailable()")
4         @ensures("entity.isBusy() && entity.isAnnounced()")
5         @advisedBy("SafetyMonitor")
6         public void move(RailEntity entity){...}
7 }
8
9 public aspect SafetyMonitor {
10
11        @requires("entity.available() && entity.isMonitored()")
12        @ensures("(entity.hasTrainBounded()?true:proc)")
13        void around(RailEntity entity) :
14                call(void *Train.move(*Entity)) && args(entity)
                      {...}
15
16 }
17
18 public class Main {
19
20     public static void main(String[] args){
21                RailEntity pe = new PlatformEntity();
22                pe.setAvailable(true);
23                pe.setMonitored(false);
24                Train t = new Train();
25                t.move(pe);
26     }
27 }
```

Figure 2.7: @advisedBy clause

kind belonging to the advised element. For non-ASP advice this element will be the next advice listed in the `@advisedBy` clause, while for ASP compliant advice it will be the advised method. This mechanism makes it possible to keep contracts reasonably compact and allows the developer to focus on the current contract.

2.4.2 Relevant subtyping issues

Another issue discussed in this work is how `call` and `execution` primitives and class hierarchies can inadvertently cause contract violations. These primitives cause pointcuts to match not only the declared type, but also its subtypes. When the dynamic type of an object is considered in a pointcut, it is safe to substitute that type for any of its ancestors, assuming that these types satisfy the strong behavioural subtyping rules. However, in pointcuts where the static type is considered, it is not assured that contracts will not be broken. Not even when this subtype and the advice behave as behavioural subtypes of

the specified class.

The programmer should only be aware of the constraints placed on the type used in the pointcut, however, in order to ensure that no problems will arise from advising a subtype at runtime it is necessary to previously check that the advice behaves as a SBS of all the types that could be advised. That is, all the types within the hierarchy. Although complex, this work could be mitigated by tools such as the one described in this thesis, setting the programmer free from that task.

CHAPTER 3

Statically ensuring modular reasoning

3.1 Transitioning to static

The dynamic algorithm mentioned in Section 2.4 makes it possible to enforce modular reasoning in ContractAJ at runtime [18]. The drawback of this procedure is that, since contracts are checked at runtime, it is necessary to execute the software to know its result, something that can be unpractical sometimes. Moreover, the results obtained are only valid for a concrete execution of the program, but do not guarantee anything about other possible runs.

This chapter presents the static equivalent for AspectJ of the mentioned algorithm. Although most of the procedure is the same for ContractAJ, some important issues that would work differently in it are also considered and explained.

The requirements that an AOP system must fulfill in order to achieve modular reasoning are the same as the ones stated by Molderez and Janssens [18], as they are due to its nature, and not dependent of how they are checked. These requirements are:

- All classes should comply with the SBS rules, taking into account the contracts of its supertype. This includes inheriting advised classes as well as overriding advice, which is possible in ContractAJ.
- To remain oblivious of an advice, it should comply with the ASP, making

all the contracts of all the advised join points hold. If a joint point is shared with a non-ASP compliant advice, it must have a lower precedence than it.

- Advice not complying with the ASP must be listed correctly in the `@advisedBy` clause.

Instead of asserting rules regarding contracts as the program executes in order to know if the ASP rules described in Section 2.4 are satisfied, it is possible to check restrictiveness of the contracts. Just this is enough to know whether an advice behaves as a behavioural subtype of another method (or advice). In case it does, it can be overseen by the rest of advice, as the behaviour of the method as is perceived by them will not change. Advice not complying with the ASP will be included in the `advisedBy` clause and treated differently, as will be explained later on.

3.1.1 Selection of join points

The set of join points selected by each pointcut must be found processing the code in order to detect the join point shadows within it. This analysis and the weaving of the aspects into the code can be performed while compiling or through bytecode manipulation but it always takes place prior to the execution of the program. In AspectJ, the weaver holds all this information and it can be retrieved through an API. In any other languages or frameworks, the weaver must provide similar features to make the static analysis of the code possible . Otherwise, an analyzer should be built.

Performing this analysis statically means that, if a pointcut contains conditions that can only be determined at runtime such as `if`, it is not possible whether it will actually be executed on each one of the related shadow points found. Although it is possible that advice defined as non-ASP by its specification behaved according to the ASP due to the conditions necessary for it to be run, only the formal specifications defined in the contracts would be taken into account.

3.1.2 Classes and inheritance

As specified in Section 3.1, all classes should satisfy the SBS rules. In ContractAJ this also affects aspects, as they are represented as regular clases and so, their advice can be overridden too. It is possible to check whether the annotated classes comply with the LSP by checking the restrictiveness of their contracts.

Besides that, there is another inheritance issue affecting pointcuts. Developers writing contracts should only take into account the static type specified within its pointcut, however, it is possible for subtypes to be advised too. A pointcut using a `call` primitive could still match cases in which the dynamic type would correspond to a subtype. The same would happen with an `execution` primitive, which would make the pointcut select the execution of any instance overriding the method declared in the specified type.

Even if the advised hierarchy complies with the LSP, it is not guaranteed that the ASP will be satisfied for all of the types within it, as explained in Section 2.4.2. For this reason it is necessary to check if the advice will satisfy the ASP principle for each of them. As this is a burden for the developer, the objective of this tool is to perform this check. This can be solved checking if the ASP holds for each type within the hierarchy advised in the pointcut that uses `execution` or `call` primitives.

3.1.3 Preconditions and postconditions

As shown in Section 2.3.2, contracts can be attached to advice as annotations. Preconditions and postconditions are associated by means of `@requires` and `@ensures` annotations, respectively. The latter adds some special keywords such as `\return` and `\old`. `\return` represents the value returned by the method and `\old` is an operator that, when used on a variable, represents the value of the referred variable before the method was executed. The use of these language extensions within contracts is common among behavioural interface definition languages.

There is another feature that can be used in preconditions and postconditions attached to around contracts. This is the `proc` keyword, which refers to the contract of the advised method. The reason why it can only be used in around contracts is that advice contracts should only concern the code within its body and around advice are able to explicitly call `proceed` in their body.

The degree to which advice contracts can be checked depends on their kind and the available information. Advice can be considered to have two boundaries, an external and an internal part. The external part would be the one associated to the contract or specification that refers to the environment where the original method was meant to be executed. Thus, the advice's external contract can be thought as a replacement of the advisee's contract. The internal part, however, stays between the execution of the advice and the advised method and its contract refers to the advice instead of to the joint point environment. In this spot is where in ContractAJ would take place the implicit

proceed call.

The contracts of advised methods always refer to their joint point environment so they can always be considered as external contracts. An around advice would wrap both preconditions and postconditions of a method, substituting them both from an external point of view, which means that its contracts would also refer to the environment and already include the behaviour of the implicit proceed call. For this reason they would be external contracts.

An example of both kind of contracts is shown in Fig. 3.1. As can be seen, the postcondition of a before advice and the precondition of an after advice are internal. As external contracts cover the same domain, both can be compared. Internal contracts, however, lie in the bond that exists between the state left by one of the elements, advice or advisee, and the one taken by the other. The check of these internal contracts would involve not just comparing the contracts, but also checking that the contract of the element that executes later is going to hold. As contracts in aspects only address their own behaviour, to know this it would be necessary to have the record of the variables that could have been changed.

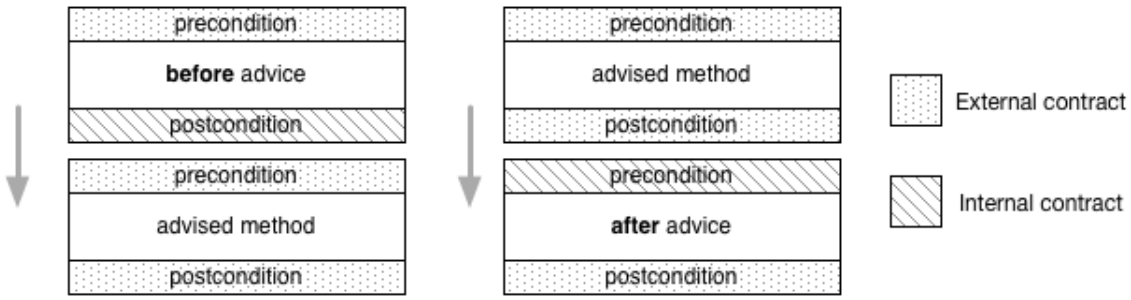

Figure 3.1: Internal contracts in before and after advice.

As this would only be possible by checking the code or with frame conditions indicating the changes made to the environment, such as the `@assignable` clause in JML [12], it is outside of the scope of this work.

Therefore, only external contracts are checked. Which contracts are external or internal depends on the kind of advice:

Before: Advice's precondition and advisee's precondition.

After: Advice's postcondition and advisee's postcondition.

Around: Advice's pre/postcondition and advisee's pre/postcondition.

3.1.4 Invariants

Invariants present a problem that does not occur with preconditions or postconditions. The behavioural subtyping rules in OOP do not always guarantee sound modular reasoning when invariants are involved [14]. This happens because these conditions must hold during all the execution of the method and that means that for practical purposes, they can be conjoined to the precondition and postcondition, altering their strength. A method could strengthen its invariant, what according to the SBS rules would be right, but at the same time it would be strengthening its precondition too, breaking the principle.

There are some restrictions that can be placed on invariants in order to make it safe to compare their strength. One would be making the variables used in the invariant private, to make impossible for a client to falsify them. Another would be, if invariants are tightened, it can only be done with constraints that were already imposed by the precondition of the method. A constraint can be understood as each one of the restrictions that a contract places over a variable and that can be extracted from such contract.

Fig. 3.2 displays an example where the contracts of a method m of a type t are expressed by means of a set of constraints obtained with the *const* function, which returns the constraints of a contract. It also shows how the constraints of the precondition (pre) and the invariant (inv) are conjoined as the union of both sets (pre_{conj}).

$$
\begin{aligned}
&const(pre(t, m)) = \{X, Y, Z\} \\
&const(inv(t) = \varnothing \\
&const(pre_{conj}(t, m)) = \{X, Y, Z\}
\end{aligned}
$$

Figure 3.2: Constraints affecting a method $t.m$.

Fig. 3.3 shows the constraints of an inheriting type u. Another constraint is added to the invariant, however, this does not strengthen the conjoined precondition as it was already imposed by the precondition of the superclass. For this reason the conjoined precondition is then weaker than the one in the superclass, complying with the SBS rules.

In Fig. 3.4 the constraints of a method $u'.m$ are displayed. As can be seen, it introduces a constraint that was not present in the subclass. This complies

$$
\begin{aligned}
&const(pre(u, m)) = \{X\} \\
&const(inv(u)) = \{Z\} \\
&const(pre_{conj}(u, m)) = \{X, Z\} \\
&const(pre_{conj}(u, m)) \subset const(pre_{conj}(t, m))
\end{aligned}
$$

Figure 3.3: Constraints on method m of a subtype u.

with the SBS rules, as the invariant is stronger, what means that is being preserved. However, introducing a new variable that was not in t causes the composed precondition of the method $u'.m$ to be stronger than the superclass, thus not behaving as a SBS.

$$
\begin{aligned}
&const(pre(u', m)) = \{X\} \\
&const(inv(u')) = \{W\} \\
&const(pre_{conj}(u', m)) = \{X, W\} \\
&const(pre_{conj}(u', m)) \not\subset const(pre_{conj}(t, m))
\end{aligned}
$$

Figure 3.4: Constraints on method m of a subtype u'.

This is an insight of how invariants could be checked but, as other complex requirements involving their ownership which are not related with the topic treated in this thesis are needed for that, their comparison goes beyond the scope of this thesis.

3.1.5 advisedBy clause

Apart from the ASP, there also is the `@advisedBy` clause which aims to restore modular reasoning in cases where the ASP cannot be satisfied.. It achieves it by listing the advice that do not comply with the ASP so that any use of the annotated method happens being aware that these advice can alter its expected behaviour.

If both ASP and non-ASP compliant advice share the same joint point, ASP advice should have lower priority. Complying with the ASP means that the advice will behave according to its expected result when advising the method or another ASP advice, as this should make no difference according to the rules of behavioural subtyping. For this reason, executing an ASP advice interleaved with ASP advice, this is, between the execution of advice listed in

an `@advisedBy` clause, could not have the desired result. The ASP advising a method can be unattended, considering only such method, as they will behave as a SBS of it, making possible to oversee them.

Non-ASP advice are free to choose any contracts so in order to check advice listed in the `@advisedBy` clause, it is necessary to confirm that they ensure the precondition of the next advice to be executed. Since advice addressing tasks such as authentication or safety, as the example shown in Fig. 2.5, have as objective halting the execution of the subsequent operations, advice being checked cannot be asked to always provide a proper environment for the next one (which is the reason why non-ASP advice must exist). Successfully checking these contracts would mean finding an execution flow that makes the contracts of the next advice hold, making them reachable. However, this is outside the scope of this thesis.

The advice listed in the `@advisedBy` clause must be listed in the expected execution order. This order is defined by the `declare precedence` statements that the developer may have used to explicitly define a safe order.

In a language that makes it possible to override advice, like ContractAJ, it is possible to reference the static type of the advice in the clause and, as long as the overriding advice is a SBS, the previously described terms are still valid in cases in which the listed advice is substituted at runtime by a subtype. However, this does not apply to this work, as there is no notion of overriding advice in AspectJ.

This clause can seem to be a burden for the developers, as they have to write its contents, partly diminishing the effectiveness of the AOP approach. Nonetheless, information contained in them is redundant and for that reason it can be inferred from processing the rest of the contracts. This makes it possible to automatically generate the clause with the contracts found that do not to comply with the ASP.

3.2 Contract strength comparison

This section explains how the restrictiveness of contracts can be used to determine whether the SBS rules are satisfied. In order to explain the comparison process, it is necessary to analyze how restrictions over the possible values of variables or methods used within in contracts characterize their strength.

3.2.1 Constraints

As briefly explained in Section 3.1.4, as contracts are defined as logical expressions, they can be rephrased into logic formulas in conjunctive normal form (CNF). For each contract $con(t, m)$ (pre, post or invariant) of a type t and a method m, there exists another contract $con_{nf}(t, m)$ specified as a CNF logical formula equivalent to it. As shown in (3.1), each one of the clauses conjoined in con_{nf} is what we refer to as a constraint (C_i). As contracts define the interaction with the environment as the expected values that certain variables must take, each one of these constraints define a restriction over those variables which is expressed as an atomic formula. Because all of these constraints are conjoined, it is necessary to satisfy all of them in order to make the contract hold.

$$con_{nf}(t, m) = C_0 \wedge \cdots \wedge C_n \tag{3.1}$$

As shown in (3.2), it is possible to define a predicate *const* as the set of the constraints C from (3.1).

$$const(con(t, m)) \equiv const(con_{nf}(t, m)) = \{C_0, \ldots, C_n\} \tag{3.2}$$

An example of a Java contract transformation can be seen in Fig. 3.5. The relational operators from the contract are transformed to atomic formulas (also called atoms), e.g. $>(6, var)$, which have no subformulas and are evaluated to a boolean value depending on their variables.

$$\begin{aligned}
&con(t, m) = 6 < var < 10 \;\&\&\; available \\
&con_{nf}(t, m) = >(6, var) \wedge <(var, 10) \wedge available \\
&const(con(t, m)) = \{>(6, var), <(var, 10), available\}
\end{aligned}$$

Figure 3.5: Example of transformation of a Java contract into constraints

Bauer et al [21] define the composition of contracts, which is represented with the operator $\otimes$ and takes two contracts as operands, in the following way:

> *A contract C is contract composition of the contracts C_1 and C_2 if C dominates C_1 and C_2[...]*

And domination of one contract over another is understood as:

> *Let C, C_1, and C_2 be contracts. C dominates C_1 and C_2 if [...] any composition of correct implementations of C_1 and C_2 results in a correct implementation of the contract C.*

Although these are defined for the particular specification theory that can be found in their work, it is possible to define such operators within our approach and not only for contracts, but also for constraints. Therefore, composition can be applied to contracts ($\otimes : con \times con \mapsto con$) and to constraints ($\otimes : const \times const \mapsto const$). Domination can also be understood among constraints in the following way: given the constraints $const$, $const_1$ and $const_2$, $const$ dominates $const_1$ and $const_2$ if the composition of $const_1$ and $const_2$ being both satisfied, satisfies $const$ as well.

3.2.2 Comparison

As seen in previous sections, the ASP is defined as relations between the different contracts that can be attached to an advice and its advisee. These relations are specified in terms of restrictiveness, that is, how difficult it is to make a contract hold. Since these relations are exclusive, it is enough to define just one of them. The chosen in this work will be "more restrictive (stronger) than", as the others can be expressed in terms of it. There are several approaches for finding out which contract out of two is stronger.

The first step is common to all of these approaches; it consists of normalizing the two contracts to be compared. This means unifying the name of the advisee variables that could have been renamed in the pointcut or the advice. The second phase would be to try and find a combination of values that satisfies one contract while the stronger one does not hold. In this point is where the path forks depending on the granularity of the element considered, i.e., depending on whether each constraint obtained from the contract or the contract as a whole is checked to be satisfied.

For the first possibility, where whole contracts are considered, the process would work as follows. An SMT Solver allows to look for a set of values for the contract variables that makes it possible for one contract not to hold while the other one does. The existence of this combination would prove that the ASP is not satisfied, and for that reason an scenario in which the ASP is not satisfied is likely to happen. Considering two contracts, one being used as a reference and the other being the target of the comparison, one of the ways to reify these values consist of asserting the reference contract, so that their restrictions are always met, and try to find a model that does not satisfy the target contract.

Another possibility is composing both contracts into one formula, as shown in (3.3). As in the other case, this presents a satisfiability problem, although it can be solved in just one step. Finding this formula to be satisfiable would mean that the target contract can evaluate to false in a situation in which the reference contract (the one visible for the developer) is satisfied. This formula applied to the contracts of each kind of advice is the key to tell whether a reference contract is stronger than a target contract and is the way in which contract comparison is implemented in the proof of concept developed.

$$cont_{ref} \wedge \neg cont_{target} \tag{3.3}$$

As explained in Section 3.1.3 this procedure can be done for each one of the contracts in the type and the supertype that need to be compared, swapping the operands if the rule demands to evaluate whether the contract is weaker. When applying this formula to each kind of advice, we obtain the following:

Before advice:

$$pre_{method} \wedge \neg pre_{advice} \tag{3.4}$$

After advice:

$$post_{advice} \wedge \neg post_{method} \tag{3.5}$$

Around advice:

$$pre_{method} \wedge \neg pre_{advice} \tag{3.6}$$
$$post_{advice} \wedge \neg post_{method} \tag{3.7}$$

The comparison of two contracts can also be made at a constraint level. In that case, the possibilities involving the two constraint sets, $const_{ref}$ and $const_{target}$, are:

- The constraints of the target are a subset of the reference subset.

  $const_{target} \subset const_{ref}$

  This is the basic case and would mean that $const_{target}$ is satisfied in its entirety if $const_{ref}$ is satisfied for all its constraints.

- Each constraint in $const_{target}$ is contained in $const_{ref}$ or is dominated by one of its constraints.

  $\forall C_1 \in const_{target} \exists C \in const_{ref}, C_2 \mid C_1 = C \quad \vee \quad C_1 \otimes C_2 = C$

  This more flexible case allows a constraint not to be directly contained in the reference set and to be dominated instead. This means that the restrictiveness of a constraint can be lower than its dominating constraint, as the latter being held implies that the dominated one will be as well.

If none of these situations happens, the contract checked as target is stronger than the one acting as reference due to the introduction of constraints that are not necessarily satisfied in the reference contract. The result of this test is the same as previously shown using the whole contracts for the comparison. However, being able to differentiate the constraints present in a formula, allows to identify the restrictiveness increments. Additionally, it makes the approach possible for invariants where it is necessary to know if a constraint was already imposed.

3.3 Java to SMT-LIB transformation

To check the satisfiability using a SMT Solver as previously explained, we chose to transform Java contracts into the common language that most SMT solvers are able to understand, SMT-LIB. The solver will evaluate the formula and check if there is a valid model or a combination of values for the variables that makes it hold, that is, if the formula is satisfiable. The transformations specified in this section use the ANTLR4 Java grammar [1]. This transformation is expressed as a function t, which takes a Java expression as input and produces an SMT-LIB expression. The following subsections define the t function for each one of the Java constructs considered.

3.3.1 Literals

Literals contain concrete values, so these values can be transformed to SMT-LIB without much effort. They can be integer and real numbers, logical constants (true and false), character and strings. For sake of simplicity, in cases in which a literal kind is a subset of another kind, the kind taken is the most abstract one. This means that integer numbers are transformed into real numbers and single characters are transformed into strings with the character as their single content. Fig. 3.6 shows this transformation.

[1] https://github.com/antlr/grammars-v4/blob/master/java/Java.g4

$\mathbf{t}(IntegerLiteral) = IntegerLiteral.0$
$\mathbf{t}(FloatingPointLiteral) = FloatingPointLiteral$
$\mathbf{t}(BooleanLiteral) = BooleanLiteral$
$\mathbf{t}('SingleCharacter') = "SingleCharacter"$
$\mathbf{t}(StringLiteral) = StringLiteral$

E.g.: `'c'` $\rightarrow$ `"c"`
`true` $\rightarrow$ `true`

Figure 3.6: Literal transformation.

3.3.2 Variables

Identifiers

Java identifiers meet the requirements for SMT-LIB identifiers so all of them can be kept without any transformation, as shown in Fig. 3.7.

$\mathbf{t}(Identifier) = Identifier$

E.g.: `var` $\rightarrow$ `var`

Figure 3.7: Identifier transformation.

Methods

Only pure methods, those which do not produce any side effects [13], are allowed to be used within contracts. Otherwise the evaluation of behaviour specifications could have effects on the software being assessed.

The way to process method calls is by substituting them with SMT-LIB variables that the solver will evaluate to all the possible values that the method can return. To achieve this, the behaviour of the method has to be defined, and this can be achieved through the assertion of the method's specification being equal to this method variable. Since the execution of pure methods should be composed of basic Java expressions and they would not have any side effects, it is possible to take the body of these methods with their instantiated parameters as their specification.

Statically evaluating these methods is not possible, as some of the variables or other methods used in them can only be known at runtime. The objective of the assertion is not to calculate the expected return of the method, but to

restrict its possible return value according to the specification. What makes this approach valid is that the asserted specification is valid for all the call instances, allowing the variables to reify to the same values and the value mapping to be consistent across them.

Fig. 3.8 shows the two main steps performed. First, the call to the method is substituted by a variable unique for that combination of method and parameters. Then, the variable is asserted to be equal to the expression returned by this method with its parameters already substituted with the values provided in the call.

$$\mathbf{t}(t.m(x_1 \dots x_n))=res_{t.m(x_1 \dots x_n)}; \mathbf{assert}(res_{t.m(x_1 \dots x_n)} = \mathbf{return}(t.m(x_1 \dots x_n)))$$

E.g.: `getA()` $\rightarrow$ `res_getA`
`assert(res_getA=a)`

Figure 3.8: Method transformation

Another way to partially solve this is to define methods as function symbols. This would require to explicitly define the result in function of their parameters. Although this might seem correct at first sight, and it would be for domains such as logic, for others (such as integer operations) it would require defining a huge amount of predicates, so it is easier just to introduce those variables into the model through an assertion, as mentioned, and let the solver evaluate them.

3.3.3 Operators

Array access

Values within arrays are referred through two elements. The first of them is the array identifier and the second one is the expression inside brackets that indicates the position being accessed. The contents of the array can only be known at runtime and the evaluation by the SMT solver of the expression regarding the element accessed would not give any additional information, as both values are not related.

The solution for this is to define a composed identifier that refers to these two elements. The first part would correspond to the array being accessed and the second part would be an identifier obtained from the index expression. Identifying the expression leads to be able to consider each element on the array and to match accesses to the same element. An example can be seen in

Fig. 3.9.

$$\mathbf{t}(Identifier[expression]) = Identifier\mathbf{f}(expression)$$

E.g.: `vector[x+2]` $\rightarrow$ `vector_id_sumx2`

Figure 3.9: Array access transformation.

Member access

The same reasoning made for vectors can be applied to the access to class members. Transforming them with the object name prefixed to the substitute variable makes it possible to unambiguously refer to each one of them. This is valid for regular attributes but also for methods, as shown in Fig. 3.10.

$$\mathbf{t}(Identifier.expression) = Identifier\,\mathbf{t}(expression)$$

E.g.: `Foo.a()` $\rightarrow$ `Foo_res_getA`
`Foo.bar` $\rightarrow$ `foo_bar`

Figure 3.10: Member access transformation.

Binary operators

Logic and arithmetic operators use an infix notation in Java, while in SMT-LIB they are prefix and are always wrapped with parentheses. This requires switching the order of the operands as well as adding the parentheses if they are not present. All the binary operators that can be used in Java can be replaced by an SMT-LIB equivalent, keeping the meaning that they had, as shown in Fig. 3.11.

Unary operators

The `!` operator becomes `not` and unary arithmetic operators are processed by applying the transformation function to a more verbose Java expression. For post-increment or post-decrement operators, only the affected variable is considered, as the value is changed after the contract evaluation, with no effect on it. Pre-increments or pre-decrements are processed by applying the transformation function to a more verbose Java expression. These concrete transformations can be seen in Fig. 3.12.

$\mathbf{t}(expression_0\ op\ expression_1) =$
$(\mathbf{t}(op)\ \mathbf{t}(expression_0)\ \mathbf{t}(expression_1))$
$\mathbf{t}(\&\&) = and$
$\mathbf{t}(\ \|\ ) = or$
$\mathbf{t}(\ ==\ ) = =$
$\mathbf{t}(!=) = not\ =$
$\mathbf{t}(\ +\ ) = +$
$\mathbf{t}(\ -\ ) = -$
$\mathbf{t}(\ *\ ) = *$
$\mathbf{t}(/) = /$
$\mathbf{t}(\%) = mod$
$\mathbf{t}(\ <\ ) = <$
$\mathbf{t}(\ >\ ) = >$
$\mathbf{t}(\ <=\ ) = <=$
$\mathbf{t}(\ >=\ ) = >=$

E.g.: `true && false` → `(and true false)`
`6 % 3` → `(mod 6 3)`

Figure 3.11: Binary operator transformation.

Ternary operators

As unary operators, Java's ternary operator `?` can be substituted by a more expressive statement such as an if-else expression. Since this expression is not accepted inside of a regular boolean expression, the transformation is directly made to SMT-LIB as shown in Fig. 3.13.

$\mathbf{t}(!expression) = (not\ \mathbf{t}(expression))$
$\mathbf{t}(expression\ ++) = \mathbf{t}(expression)$
$\mathbf{t}(expression\ --) = \mathbf{t}(expression)$
$\mathbf{t}(\ ++expression) = \mathbf{t}(expression + 1) = (+\ \mathbf{t}(expression)\ 1)$
$\mathbf{t}(\ --expression) = \mathbf{t}(expression - 1) = (-\ \mathbf{t}(expression)\ 1)$

E.g.: `!true` $\rightarrow$ `(not true)`
`6++` $\rightarrow$ `(+ 6 1)`

Figure 3.12: Unary operator transformation.

$\mathbf{t}((expression_0\ ?\ expression_1 : expression_2)) =$
$(and\ (or\ (not\ expression_0)\ expression_1)(or\ expression_0\ expression_2))$

E.g.: `(6>2?true:false)` $\rightarrow$
`(and(or(not(>6 2)true)(or(> 6 2)false))`

Figure 3.13: Ternary operator transformation.

CHAPTER 4

ASP Checker

A tool able to check the concepts explained in Section 3 was developed and is available in an online repository[1]. Since not all concepts are implemented, the tool possesses some limitations that will be described later on.

4.1 Plugin design

The tool developed is a plugin for AJDT (AspectJ development tools)[2]. AJDT is an Eclipse Platform based tool that supports AOP with AspectJ and aims to deliver a user experience that is consistent with the Java Development Tools (JDT) when working with AspectJ[8].

To be able to statically retrieve information from AspectJ programs, we make use of the Ekeko meta-programming library[3]and its and its extension GASR (General-purpose Aspectual Source code Reasoner)[4] [20]. Written in Clojure, Ekeko makes it possible to query and manipulate the Java/AspectJ source code in Eclipse projects, providing an API and an interactive networked REPL (Read Eval Print Loop) shell. Instead of developing from scratch some of these tasks that would be difficult to implement, the plugin relies on Ekeko.

[1] `http://github.com/rapsioux/asp-checker`
[2] `http://www.eclipse.org/ajdt/`
[3] `http://github.com/cderoove/damp.ekeko`
[4] `http://github.com/cderoove/damp.ekeko.aspectj`

The following sections discuss these main tasks along with the design options considered and the decisions finally taken. Fig.4.1 shows a general diagram of how information flows within the modules that interact with the plugin.

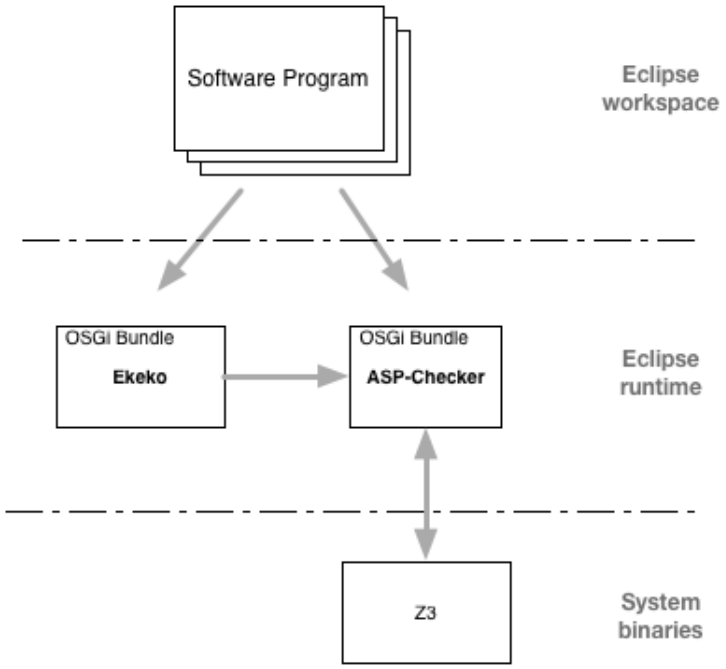

Figure 4.1: Overview of the tool.

4.1.1 Contract specification

To be able to compare the contracts it is necessary to somehow attach them to the source code that has to be analyzed. Java provides annotations as a simple way to mark elements with custom information. These annotations can later be processed with an annotation processor, through runtime reflection or parsing and analyzing the code.

4.1.2 Model data retrieval

AJDT supports gathering the existing elements from the code like aspects, advice, pointcuts and shadow join points. It also makes it possible to get the relationship between this elements, such as the shadow join points generated by a certain pointcut or the methods advised by a specific advice.

The API can be used from any AJDT plugin, however, Ekeko provides the same functionality and allows to avoid, to a certain degree, the internal AJDT and JDT classes. Ekeko is written in Clojure, which makes it possible to use functional programming to simplify some tasks.

Using Clojure implies having to use a Clojure OSGi loader in order to run it from an Eclipse environment and run the code from a REPL.

4.1.3 SAT Checking

To check the satisfiability of the formulas seen in Section 3.2 it is necessary to use of a SAT Checker or an SMT solver.

SAT solvers allow us to specify a clause in a conjunctive normal form (CNF) containing predicates and determine its satisfiability. SMT solvers are capable of even more, accepting clauses that are not normalized. Moreover, they already implement some of the features regarding interpretation into specific domains such as Integers or Arrays. These recognized domains are known as theories and can be useful when working with Java code, as they define part of it. They allow the specification of functions and custom sorts as well.

For this reason, SMT solvers are easier to adapt to model a programming language and to use in this context than SAT solvers. From all the SMT solvers curretly available, such as Z3, Yices or CVC4, Z3 was chosen due to its larger community support, the maturity of the project and the ease of using it from a Java Runtime Environment.

4.1.4 Contract parsing and transformation

After gathering all the contracts from the code, it was necessary to transform them into the notation that most SMT solvers, including Z3, use: SMT-LIB.

Transformation languages such as TXL[3] or Rascal[9] work by describing the grammar rules that have to be applied in order to get a language converted to another one. This approach is simple yet powerful, however, it is necessary to identify the sort of the Java identifiers in order to feed them to the SMT Solver and that requires an intermediate step of semantic analysis[1].

Taking into account these requirements, the option chosen was to use a parser with a provided grammar to perform the mentioned analysis on the Parse Tree.

The process to parse the input and get the type or "sort", as they are called

in the context of SMT solvers, of the variables is split in two phases. The first of them proceeds as follows.

A parse tree is generated for each contract. Then, a transformation rule is applied to the root of the tree. Depending on the element found, the current rule is transformed and the children nodes that form part of it are recursively visited. The transformation can be:

- For parentheses expressions, the parentheses are removed.
- For logic, arithmetic, relative comparisons and equality expressions, the infix operator is converted into a prefix position.
- For unary and ternary operators, these are decomposed into a more verbose expression. E.g. `++x` to `x+1`.
- For terminal nodes, such as literals and identifiers, these are returned without any action.

In the second step, the parse tree is visited following a pre-order traversal. For each subtree that generates a SMT-LIB variable (Java identifiers), the sort is inferred and added to a hash map that contains the identifier as key and the sort as value.

Inferring the sort of a SMT-LIB variable derived from a rule is done as follows. There are two ways to find the sort depending on how the parse tree is visited, ascending and descending, and depending on the context one or the other can be used. In the same manner as the contract transformation, the parse tree is recursively visited and, as soon as the type is found out, the process stops.

The default deduction method is ascending recursion, and works in the following way:

- If the rule is a top level contract expression, the sort is `bool`.
- If the immediately superior rule takes logic operands, the sort is `bool`.
- If the rule at a higher level is an arithmetic expression or a comparison expression, the sort is `real`.
- In case of the expression above being an equality expression, the sort is found out by descending on the equaled term.

Descending recursion works in a similar way to the previous schema but the matched type corresponds to the resulting type of the literal or expression found on the child nodes.

An example of type inference on a simple contract is shown in Fig.4.2. The sort of the method `available()` can be retrieved using the rule above it, as the operands of a logic *and* expression must be bool. The variables `accepted` and `state`, however, belong to an equality clause, so the type must be retrieved from the other side of it. As can be seen, this assigns `accepted` the sort bool and `state` the sort real.

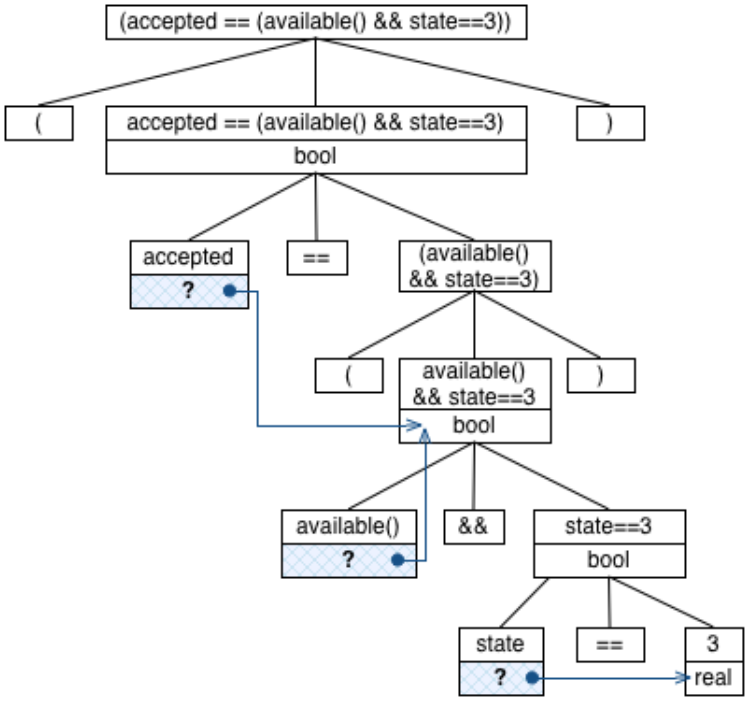

Figure 4.2: Type inferring from the parse tree.

Java identifiers used within the contracts are not checked against the corresponding members or parameters outside them. Although this would ease the task of retrieving the sort for each one, such checks are complicated by features such as intertype declarations and parameter bindings.

However, the approach taken is valid provided that the names of the variables are not different between the advice and the advised method. The results obtained from the verification using the SMT Solver are equally valid, being

input validation the only point neglected.

Furthermore, method calls as well as vectors, explained in Section 3.3 are not supported in the current version. Due to the way in which they are implemented, they will be overlooked and interpreted as a plain variable. Although this may not cause any problems for vectors, for method calls some false positives could be triggered. This happens because the internal behaviour of the method is not taken into account and two methods that must return the same value can be reified to different values. Therefore, method calls will have undefined behaviour and should not be used in contracts.

4.2 Usage of the tool

4.2.1 Adding the necessary metadata

To be able to check contracts in the ways described before, it is necessary to add some additional information to them. This information is added as Java annotations.

Annotations should be exported as documented in the installation manual available in the repository and added to the classpath of the project in which the metadata is needed. All of them accept a single String parameter and they can be stacked, each one being present at most once in each suitable element. They are:

- `@requires`
- `@ensures`
- `@invariant`
- `@advisedBy`
- `@name`

Three kind of annotations can be distinguished depending on the elements that they affect. The first one encompasses `@requires`, `@ensures` and `@invariant`.

The parameter for these, representing a contract, must be a Java boolean expression, using the variables visible to the element they are attached to.

They can apply to regular class methods and to the advice that advise them. The corresponding annotation should be placed in both the advised method and the advice. One of them not having an annotation while the other does will trigger an error, which is further explained in Section 4.2.2. The annotations retrieved do not necessarily need to be of the same kind, but of the kinds necessary to check behavioural subtyping for that kind of advice. To know what contracts are combined together with that purpose, check Section 3.1.3.

Some valid examples of these annotations are the following:

```
@requires(‘‘param==2 && inState==false’’)
@invariant(‘‘constant==\"value\"’’)
@ensures(‘‘true’’)
```

The second kind of annotation has a different behaviour: `@advisedBy`. It only applies to methods and takes as an argument a sequence containing the names of the advice that advise the associated method in the form `@advisedBy(‘‘[advice-name]...’’)`. It is only required for advice known to break the ASP.

Advice contained in an `@advisedBy` clause are expected to be executed in the order declared within the clause. If another order has been explicitly declared in an aspect via the `declare precedence` directive, a warning will be thrown, as the runtime execution order could be different than the specified by the clause and have unexpected results. For more information about this error, see Section 4.2.2.

Valid `@advisedBy` uses are:

```
@advisedBy(‘‘LogGetter’’)
@advisedBy(‘‘LogGetter ChangeListener CheckRestrictions’’)
```

The last annotation type is `@name`. It applies to advice and makes it possible to give them a name, something required as seen in Section 3.1.5. Takes a single argument that corresponds to the name given to the advice.

Use examples:

```
@name(‘‘LogGetter’’)
@name(‘‘CheckRestrictions’’)
```

4.2.2 Running the tool

As explained in Section 4.1, the tool is implemented as an Eclipse plugin that works on top of Ekeko.

The steps to run from Eclipse are:

1. Enable the Ekeko nature for the projects that have to be analysed. This can be done through the package explorer, the context menu of each project, and selecting the option *Configure* and then *Include in Ekeko Queries*.
2. In the menu *Ekeko*, select the option *Start nREPL* and take note of the address.
3. In the menu *Window*, select *Connect to REPL* and input the adress from the previous step.
4. Load the checker's workspace and run it:

   ```
   (use 'be.ac.ua.aspchecker.core)
   (run)
   ```

The results of the performed check should appear shortly in the REPL. All the issues that need to be reviewed are printed to the shell. An empty execution trace, without any error message means that there are no elements that require the user's attention.

Note that after each message, a `nil` will appear due to how messages are printed in the REPL. It is only the result of the printing operation and does not carry any meaning.

Moreover, at the time of writing this thesis, due to an existing bug in the weaver API exposed to Ekeko, the latter returns duplicated results for queries and, for that reason, so are the results obtained from this plugin.

If the advice is not a behavioural subtype of the advised method, the `@advisedBy` clause is checked. Not mentioning the said advice in such clause would mean that the ASP is broken, and an error is printed for that combination of advice, method and contract.

Printed messages regarding violations of the ASP always contain this information:

1. Message describing the issue found.

2. Signature of the method.
3. Name of the advice if present.
4. Location in the source code of the advice that advises the mentioned method.

An example output for one of this messages is:

```
Around advice:the invariant cannot be weakened.
int getX()
LogGetter
/src/ua/ac/ab/Logger.aj:23
nil
```

If some of the annotations required to check the ASP of a pair method-advice, but not all of them are present in the code, a warning will be issued containing the previous information and with the next message:

```
"Ensure that contracts exists on both, advice and advised method."
```

To fix it, add the required contracts, specified in Section 3.1.3 or remove those already placed.

Another check performed is the order verification within the `@advisedBy` clause. If the advice advising a method can be executed in a different order than the one specified, an error with the said information and the next message will be issued:

```
"Advice in advisedBy clause are not necessarily executed in that
order."
```

The function `run` executes all these checks, however, they can also be called individually with the functions `check-asp` and `check-advisedby`, also available on the `be.ac.ua.aspchecker.core` namespace.

4.3 Extending the tool

4.3.1 Process

The plugin can be easily extended in two different ways and in both cases, it should be used through the nREPL provided by Ekeko. The first one is forking

the repository[5] and adding the functionality as Clojure namespaces into the path `src/be/ac/ua/aspchecker`.

The second way is creating a new plugin that uses the ASP checker namespaces. Use the mentioned repository as a reference to develop it. There are some key points to make the plugin work.

The first one is loading the Clojure code as an OSGi module. This can be done using the class `ccw.util.osgi.ClojureOSGi` provided by counterclockwise. The module must be explicitly loaded before it is used in the interactive shell, which is done in the ASP checker by making it load each time Eclipse starts. If needed, the said example is available in the activator class of the checker[6].

The second point is registering the needed plugins as buddies in Eclipse. It can be done using the `Eclipse-RegisterBuddy` property in the `MANIFEST.MF` file. For an example, check the mentioned file of the plugin[7].

4.3.2 Using the checker features

There are five Clojure namespaces that provide the functionality to perform the checking tasks. They are into the `be.ac.ua.aspchecker` path and are:

core The entry point for the application. Rebuilds all the projects enabled for Ekeko and check the contracts for each one of them.

model An abstraction layer that uses Ekeko to query the needed data from the weaver model.

transformation Transforms a Java boolean expression into a SMT-LIB formula and retrieve the sorts.

contracts Stores how contracts from an advice and a method should be restricted and checks them.

z3 The interface to the Z3 solver. Create symbols from the specified sorts and feed it the clause.

[5] `http://github.com/rapsioux/asp-checker/`

[6] `http://github.com/rapsioux/asp-checker/blob/master/src/be/ac/ua/aspchecker/init/Activator.java`

[7] `http://github.com/rapsioux/asp-checker/blob/master/META-INF/MANIFEST.MF`

CHAPTER 5

Related work

ASP and @advisedBy clause

The work that is most closely related to this thesis is Molderez and Janssens' [18], which defines basic concepts relevant to the maintenance of modular reasoning in aspect oriented languages such as the Advice Substitution Principle and the `@advisedBy` clause. The main difference between both works is that they concretize these concepts from a dynamic point of view, asserting software contracts to check if they hold as the program executes. Instead of monitoring contracts at runtime, our static approach tries to find if there exists the possibility of that situation happening computing all the potential values that the variables within contracts can take. The drawback is that our approach leaves uncovered some contracts, as frame conditions specifying which variables are changed are needed, as explained in Section 3.1.3, while the dynamic test would succeed in these checks.

Model checking

Several authors consider addressing the same verification issues through model checking. Katz and Rashid [10] present a framework for generating proof obligations in AOP programs from the requirements. Those are then joined to others inferred from the aspect-oriented design domain. These rules can then be used to check implementation or as input for model checkers. Aldrich [2] defines a formal system that can be used to know if the behaviour of a program that uses AOP is being altered uncorrectly by comparing unadvised and advised versions of the software. Mostefaoui and Vachon [19] and Ciraci [4] provide a way of checking UML models by transforming them into graphs or

petri nets that take into account all the possible executions and check them against declarative specifications. Durr et al [5] present how composition filters can be used to detect issues derived from advice ordering. The desired behaviour is specified in form of filters, and then all the possible ordering of the advice is checked in order to know if the conditions described in the filters can be met. They have as drawback that calculating and checking all the possible runs of a software program is computationally very expensive.

CHAPTER 6

Conclusion and future work

This presented how modular reasoning in Aspect Oriented Programming can be maintained from a static point of view. Statically performing this check allows us to find all the points within a program where the execution of aspects can lead to unexpected behaviour without the need to run the program and without limiting the scope to the current execution. The work of Molderez and Janssens' is used as a basis, presenting some of the concepts introduced by them, such as the Advice Substitution Principle (ASP) and the `@advisedBy` clause but applied to this static approach. Both compliance with the ASP advice and the specification of `@advisedBy` clauses complement each other and stay as requirements to maintain modular reasoning.

It has also been described how to check compliance with the ASP by checking the rules that it imposes. As the ASP rules are described by means of preconditions, postconditions and invariants (or contracts), it is possible to compare these behavioural specifications to find out whether the ASP is satisfied. The comparison is made by checking which one of two given contracts is more restrictive and, since restrictiveness can be understood as the relative difficulty to make it hold or satisfy it, it can be translated into a satisfiability problem and solved through the use of a Satisfiability Modulo Theories (SMT) solver. It has also been presented how the contracts written in a small subset of the Java language can be translated into the language used by most of the SMT solvers (SMT-LIB).

As future work, there are some paths that could be followed in order to achieve a more exhaustive checking. One would be achieving completeness in

the transformation of Java boolean contracts to SMT-LIB, covering all the subset of the language that can be used. Another path that could be taken is finding out how to detect increments in the restrictiveness in contracts and how to apply this to successfully check invariants. Beyond that, it could also be possible to check the correctness of the code, ensuring that it behaves as the contracts state.

Regarding the software built as a proof of concept, the use of methods as it has been specified in this work could be implemented and `@advisedBy` clauses could be automatically created with the results of the ASP check performed.

Bibliography

[1] A. V. Aho and et al. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 2007.

[2] Jonathan Aldrich. Open modules: Modular reasoning about advice. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2005.

[3] Queen's University at Kingston. The txl programming language, 2013.

[4] Selim Ciraci, Wilke Havinga, Mehmet Aksit, Christoph Bockisch, and Pim van den Broek. A graph-based aspect interference detection approach for uml-based aspect-oriented models. In Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, pages 321–374. Springer Berlin Heidelberg, 2010.

[5] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Static and dynamic detection of behavioral conflicts between aspects, 2007.

[6] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.

[7] The Eclipse Foundation. Aspectj.

[8] The Eclipse Foundation. Ajdt - aspectj development tools project, 2014.

[9] Centrum Wiskunde & Informatica. Rascal metaprogramming language.

[10] S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 48–57, Sept 2004.

[11] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[12] Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *ICFEM*, pages 2–34, 2006.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[14] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, behavioral subtyping, specification inheritance, and modular reasoning. Technical report, Department of Computer Science, Iowa State University, 2006.

[15] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[16] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[17] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[18] Tim Molderez and Dirk Janssens. Modular reasoning in aspect-oriented languages from a substitution perspective. *Transactions on Aspect-Oriented Software Development*, 2014 In Press.

[19] Farida Mostefaoui and Julie Vachon. Formalization of an aspect-oriented modeling approach. In *In Proceedings of Formal Methods 2006*, 2006.

[20] Coen De Roover and Reinout Stevens. Building development tools interactively using the ekeko meta-programming library. In Serge Demeyer, Dave Binkley, and Filippo Ricca, editors, *CSMR-WCRE*, pages 429–433. IEEE, 2014.

[21] Sebastian S. Bauer et al. Moving from specifications to contracts in component-based design. In Juan Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer Berlin Heidelberg, 2012.

[22] Dean Wampler. Aspect-oriented design principles: Lessons from object-oriented design. In *Sixth International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver, British Columbia, 2007.