Expressing Design by Contract Principles using Aspects

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in Tel-Aviv University, School of Computer Science

by

Ohad Barzilay

Prepared under the supervision of Prof. Amiram Yehudai, Dr. Yishai A. Feldman, and Dr. Shmuel Tyszberowicz

July, 2004
Acknowledgments

I would like to thank my instructors. Dr. Shmuel Tyszberowicz who was always there for me from the very beginning, and contributed his effort and experience. Dr. Yishai A. Feldman for his undaunted professionalism and his uncompromising excellence. I am most indebted to Prof. Amiram Yehudai who orchestrated this research. His vision, his sincere attitude and his modesty inspire me and give me a model to look up to as a young researcher.

I would like to thank my friends and fellow graduate students - Erez Greenstein, Yaron Ostrover, Gilad Arnold and Zur Izhakian for their encouragement and support.

To my parents Amalia and Meshulam Barzilay.

And to Galia. My love.
Abstract

This thesis deals with two methodologies in software construction: Aspect Oriented Programming and Design by Contract. It was already claimed that the design by contract methodology is an aspect of the software system. As such, a contract can be expressed in AOP terminology, and hence could be enforced using an aspect oriented based tool. The work presented here describes the enforcement process, and divides it into the building blocks of design by contract. For each building block we present a constructive way to express its essence in AspectJ, an aspect oriented programming language.

The devil lies in the details. There are several issues as precondition subcontracting, collecting old values and others that needed some research effort in both AOP and DbC fields. The research yielded a discussion about AspectJ semantics on the one hand and variants of design by contract on the other. On the AOP part this work investigates the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. On the DbC part this work presents some variants and extensions to original design by contract introduced by Meyer, namely contract enforcement with respect to the client, class oriented contract (static assertions), enforcement of reflexive contracts and others.
# Contents

1 Introduction ............................................. 3
   1.1 Background ........................................... 3
   1.2 Design by Contract ................................. 4
      1.2.1 Principles of Design by Contract .............. 4
      1.2.2 Inheritance .................................... 5
      1.2.3 Notation ....................................... 5
   1.3 Aspect Oriented Programming ..................... 6
      1.3.1 AOP Principles .................................. 6
      1.3.2 The AspectJ Language .......................... 7
   1.4 Intuition ............................................ 9
   1.5 Contributions ....................................... 10
      1.5.1 Expressing Contracts as Aspects ............... 10
      1.5.2 Building a Tool ................................ 10
      1.5.3 DbC Variations and Extensions ................ 10
      1.5.4 AspectJ Semantics .............................. 10
   1.6 Related work ....................................... 10

2 Implementing DbC with AOP ............................. 12
   2.1 Motivation .......................................... 12
   2.2 Preconditions ....................................... 14
   2.3 Postconditions ...................................... 16
   2.4 Prev .................................................. 18
   2.5 Class Invariants .................................... 21
   2.6 Constructors ....................................... 22
   2.7 Interfaces .......................................... 25

3 Variants and Extensions of DbC ....................... 28
   3.1 Class Oriented Contracts ........................... 28
   3.2 The Client Perspective ............................. 29
      3.2.1 Dynamic vs. Static Contracts ................ 29
      3.2.2 Taking Client’s Perspective in Jose .......... 30
      3.2.3 As Good as it Gets ............................ 32
   3.3 Recursive and Reflexive Assertions ............... 33
3.3.1 Eiffel Solution ................................................................. 33
3.3.2 Infinite Recursion ............................................................. 34
3.3.3 The Meaning of Assertions of Assertions ............................... 35
3.3.4 Preventing Infinite Recursion in Jose .................................. 35
3.4 Precondition Implies the Postcondition Implicitly .................... 36
  3.4.1 Two Flavors of Postconditions ......................................... 36
  3.4.2 Implementing Explicit Derivation in Jose .......................... 37
3.5 Non Destructive DbC Assertions .......................................... 37

4 Further Research ................................................................. 40
  4.1 Other Languages and Methodologies ..................................... 40
  4.2 AspectJ Semantics ............................................................ 40

A Call and Execution Semantics in AspectJ ................................. 41
Chapter 1

Introduction

1.1 Background

Software construction is hard. It comes as a surprise to a beginner programmer that falsely assumes that learning a programming language is about everything. Large software systems have problems beyond source coding. Difficulties in software evolution and problems maintaining large programs are known as the software crisis, and there are several different approaches to handle those problems. (Although it is agreed that there is no silver bullet [3]). Some of the approaches address the programming language, others focus on the development process an the project life cycle.

Design by contract (DbC) and aspect oriented programming (AOP) are two of these attempts. Both of them don’t add strength to the underlying programming language, but they intend to help the underlying language in means of modularity and code quality.

The aspect oriented technology tries to improve software construction by capturing cross-cutting concerns in a modularized way. AspectJ, a general purpose aspect oriented language, offers a language construct called an aspect to capture such concerns.

Design by contract asks the code writer to add assertions to every class and every method, and to define the privileges and responsibilities (the contract) between the method and its clients. Those assertions serve four purposes: aid in constructing correct programs; documentation aid; debugging aid; basis for an exception mechanism. All these purposes attempt to improve the quality of the software.

Design by contract was introduced by Bertrand Meyer who also created Eiffel [21] — a general purpose object oriented programming language which has a runtime mechanism for design by contract enforcement. Java doesn’t have such a mechanism, and runtime enforcement can be done only by using external tools. Some tools try to augment Java with such a mechanism (such as JMSAssert [19], iContract [10, 25] and JContract [23]). Those tools mixed design by contract logic with code instrumentation. Describing the contract as an aspect is an important step when building such a tool. The reduction into the aspect oriented world simplifies some of the difficulties that arose in non-AO approaches.

Expressing design by contract building blocks in aspect oriented language is not only academic, it sets the foundations for building a software tool named Jose (when said in Spanish
it sounds like the word ‘contract’ in Hebrew) which was built using AspectJ technology. Jose enforces a system contract by generating AspectJ aspects, which in runtime, check the relevant assertions and report contract violations. This way the enforcement could be managed centrally both at compile time and at runtime. Unlike other tools, Jose does not instrument the class source code or byte code. The contract gets the plugability nature of the aspect. Jose tool proves by example the feasibility and significance of such work.

While implementing design by contract principles, we found that we can extend the principles introduced by Meyer with some of our own. In some cases we dispute the way contracts are enforced in Eiffel (a contract of a contract is not enforced). In others we found a need to add a new type of assertion (static invariant), and in third case we found two optional enforcement mechanism that the user may want to choose between (enforcement with respect to the client of to the supplier and explicit implication of the precondition of the postcondition). Chapter 3 describes the different variants and alternatives.

Generating aspect oriented code reveals some delicate issues in the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. We will present the semantic model manifested by the 1.1.1 release of AspectJ, point out its shortcomings, and present alternative models. The significance of this discussion increases as there is yet no formal semantics for AspectJ, and this work contributes to the community discussion for a “right” semantics.

The rest of this chapter provides a quick review of the AspectJ language and the design by contract principles. Chapter 2 is a detailed description of a contract as a software aspect in the AspectJ programming language. Chapter 3 presents some variants and extensions to original design by contract introduced by Meyer [21, 22], namely contract enforcement with respect to the client, class oriented contract, different treatment in reflexive contracts and others. In the appendix we discuss AspectJ’s semantics with relation to inheritance, and we present some alternative semantics and investigate their expressive power.

1.2 Design by Contract

1.2.1 Principles of Design by Contract

Design by contract is a set of principles [21] that ask the programmer to add assertions in a systematic way to her code.

Assertions are boolean expressions expressing the semantic properties of classes and reintroducing the axioms and preconditions of the corresponding abstract data types. Assertions are used in preconditions (requirements under which routines are applicable), postconditions (properties guaranteed on routine exit) and class invariants (properties that characterize class instances over their lifetime).

A precondition and a postcondition associated with a routine describe a contract between the class and its clients. The contract is only binding on the routine inasmuch as calls observe the precondition; the routine then guarantees the postcondition on return. The notion of contracting provides a powerful metaphor for the construction of correct software.

1Part of the discussion was presented in AOSD 2004 in FOAL workshop.
The invariant of a class expresses the semantic constraints on instances of the class. The invariant is implicitly added to the precondition and the postcondition of every exported routine of the class. An implementation invariant, part of the class invariant, expresses the correctness of the representation vis-à-vis the corresponding abstract data type.

Assertions serve four purposes: aid in constructing correct programs; documentation aid; debugging aid; basis for an exception mechanism.

1.2.2 Inheritance

Invariants of parents are automatically added to a class’s invariant. In the design by contract approach, inheritance, redefinition and dynamic binding introduce the concept of subcontracting. A routine redeclaration (redefinition or implementation of an abstract method) may keep or weaken the precondition; it may keep or strengthen the postcondition.

In Eiffel, an assertion redeclaration may only use require-else mechanism (for or-ing of preconditions) and ensure-then mechanism (for and-ing of postconditions). In Jose, any use of @pre and @post is implicitly considered as or-ing or and-ing assertions in parent class respectively. In the absence of these clauses the routine keeps the original assertions.

\[
\begin{array}{|c|}
\hline
C \\
\hline
\end{array}
\quad \rightarrow \quad
\begin{array}{|c|}
\hline
A & \@inv: \omega \\
\hline
r() & \@pre: \alpha \\
& \@post: \beta \\
\hline
\end{array}
\quad \rightarrow \quad
\begin{array}{|c|}
\hline
B & \@inv: \sigma \\
\hline
r() & \@pre: \gamma \\
& \@post: \delta \\
\hline
\end{array}
\]

Figure 1.1: a simple client-supplier relationship with inheritance.

1.2.3 Notation

A variant of the following notation is used by most of current Java design by contract tools. We are using this notation as well.

Figure 1.1 is a simplified UML class diagram describing a class \( C \) which is a client of class \( A \) (or interface \( A \)). By a client we mean that one of \( C \)'s data members is a reference of type \( A \) or one of \( C \)'s methods has a local variable which is a reference of static type \( A \). \( A \)'s invariant is the boolean expression \( \omega \). Along our discussion we use Greek letters names to denote boolean assertions. Such assertions are well formed Java boolean expression and may contain method calls. The client \( A \) has a member method named \( r() \) which has a precondition \( \alpha \)
and postcondition beta. By saying that alpha is the precondition of r(), we mean that before any call to r() it is C’s responsibility to ensure alpha. By saying that beta is the postcondition of r() we mean that upon any exit from the method r() when alpha held at the entry, class A guarantees that beta holds.

Class B extends (inherits from) class A. Class B has an invariant sigma. Due to the principles described earlier, the effective invariant of class B is sigma && omega. B redeclares the method r() (method overriding) and redefines the precondition to be gamma and the postcondition to be delta. Due to the subcontracting principles the effective precondition of r() is gamma || alpha (implicit require-else) and the effective postcondition of r() is beta && delta (implicit ensure-then). This also conforms with the subcontracting principles that preconditions can only be weakened and postconditions may only be strengthened [21, 22].

Note that due to polymorphism the client C may actually reference an object of dynamic type B although the static type of C’s reference is A. Keeping the subcontracting principles while enforcing design by contract assertions is crucial for such scenarios.

1.3 Aspect Oriented Programming

1.3.1 AOP Principles

One of the most fundamental principles in software engineering is the separation of concerns (SOP) [24]. SOP leads to a system that is simpler to understand and easier to maintain. Various methodologies and frameworks exist to support this principle in some form. For instance, with OOP, by separating interfaces from their implementation, one can modularize the core concerns well. However, for crosscutting concerns, concerns that cut across various modules of the system, OOP forces the core modules to embed the crosscutting concern’s logic. While the crosscutting concerns themselves are usually independent of each other, the use of OOP leads to an implementation that no longer preserves the independence in implementation.

The most common solution to the difficulties of crosscutting concerns is to develop new domain-specific solutions, such as the EJB specification for enterprise server-side development. While these solutions do modularize certain crosscutting concerns within the specific domain, their usefulness is restricted to that domain. The cost of using these pre-wired solutions is reflected in the time and effort that is required to learn each new technology that, in the end, is useful only within its own limited scope.

Aspect-oriented programming (AOP) changes this by modularizing crosscutting concerns in a generic and methodical fashion. With AOP, crosscutting concerns are modularized by encapsulating them in a new unit called an aspect. Core concerns no longer embed the crosscutting concern’s logic, and all the associated complexity of the crosscutting concerns is isolated into the aspects. AOP marks the beginning of new ways of dealing with a software system by viewing it as a composition of mutually independent concerns.
1.3.2 The AspectJ Language

The Dynamic Join Point Model

A critical element in the design of any aspect-oriented language is the join point model [28]. This model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns. AspectJ’s join points are well-defined points in the execution of the program.

AspectJ supplies many kinds of join points: method call and execution join points, constructor call and execution, field set and reference, and more.

A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception).

A method execution join point differs from a method call join point by the context of the execution. In the call join point it is the caller which is the executing object while at the execution joint point it is the object where the method is declared.

At runtime, each method call is a different join point, even if it arrives from the same expression in the program. Many other join points may be active while a method call join point is executing – all the join points that occur while executing the method body, and in those methods called from the body. We say that these join points execute in the dynamic context of the original call join point.

Pointcuts

Pointcuts match certain join points in the program flow. For example, the pointcut call(void Point.setX(int)) defines the set of all joinpoints that represent calling a method with the signature void Point.setX(int).

A pointcut can be built out of other pointcuts with and, or, and not (spelled &&, ||, and !). Pointcuts can identify join points from many different types - in other words, they can crosscut types. AspectJ allows programmers to define their own named pointcuts with the pointcut form, and whenever this definition is visible, the programmer can simply use that name to capture this pointcut.

AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. This feature is called property-based crosscutting. The simplest of these involve using wildcards in certain fields of the method signature.

The cflow pointcut identifies join points based on whether they occur in the dynamic context of other join points. cflow(<join point signature>) matches each join point that occurs in the dynamic context of the specified join point. This picks out each join point that occurs between when a method that matches the join point signature is called and when it returns (either normally or by throwing an exception).

Advice

Advice is used to actually implement crosscutting behavior. An advice brings together a pointcut (to match join points) and a body of code (to run at each of the matched join points). When a
pointcut matches a certain joinpoint, its advice is performed. AspectJ has several different kinds of advice. The before advice runs as a join point is reached, before the program proceeds with the join point. The after advice of a particular join point runs after the program proceeds with that join point. For example, an after advice on a method call join point runs after the method body has run, just before control is returned to the caller. Because Java programs can leave a join point ‘normally’ or by throwing an exception, there are three kinds of after advice: after returning, after throwing, and plain after (which runs after returning or throwing, like Java’s finally). Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point.

Exposing Context in Pointcuts

Pointcuts can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations. An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses.

The advice’s pointcut publishes the values for the advice’s arguments. The three primitive pointcuts this, target and args are used to publish these values.

A named pointcut may have parameters like a piece of advice. When the named pointcut is used (by advice, or in another named pointcut), it publishes its context by name just like the this, target, and args pointcut.

Inter-type Declarations

Inter-type declarations (also known as method introduction) in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. Unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time.

Consider the problem of expressing a capability shared by some existing classes that are already part of a class hierarchy, i.e., they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to each affected class a method that implements this interface.

AspectJ can express the concern in one place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields with the existing classes.

Aspects

Aspects wrap up pointcuts, advice, and inter-type declarations in a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers in addition to the crosscutting members. Because only aspects may include these crosscutting members, the declaration of these effects is localized. Like classes, aspects may be instantiated, but AspectJ controls how that instantiation happens – so you can’t use Java’s new form to build new aspect instances. By default, each aspect is a singleton, so one aspect instance is created. This means that advice may use non-static fields of the aspect, if it needs to keep state around.
Aspects may also have more complicated rules for instantiation, but these are not described here.

For more about the AspectJ programming language refer to the *AspectJ programming guide* [28] and the books *AspectJ in Action* [12] and *Mastering AspectJ* [7].

### 1.4 Intuition

Consider a method `m()`, that returns no value (`void`) with precondition `alpha`. The contract for this method in Java is depicted in Figure 1.2. The enforcement of the contract (for now we ignore inheritance considerations) instruments the code as shown in Figure 1.3.

```java
/**
 * @pre alpha
 */
void m()
{
  // do something
}
```

Figure 1.2: void method `m()` with precondition `alpha`.

```java
void m()
{
  if(!alpha)
    throw new PreconditionViolationException();
  // do something
}
```

Figure 1.3: Enforcing precondition using a `throw` clause.

Code instrumentation can be done at the source code level or on the compiled byte code, but it means the same. Such a wide manipulation on the code is scattered around the system and crosscut all the modules. Contract enforcement is a *crosscutting concern* of the system, and therefore an aspect. Aspect oriented programming offers a means to generalize this process.

Existing tools suffer from mixing the code instrumentation with design by contract logic and violating the separation of concerns principle. By using AOP with AspectJ we are going to present a clean implementation that is going to leave all the instrumentation work to the AO weaver (AspectJ “compiler”), and concentrate on specifying the contract in terms of aspects.
1.5 Contributions

1.5.1 Expressing Contracts as Aspects

It is not surprising that DbC methodology is an aspect of a software system. Showing that involves a full and comprehensive translation from the abstract ideas as described by Bertrand Meyer, and demonstrated in the Eiffel programming language, into the world of AOP languages. It is the first time to present such a full description that includes the subtle differences between methods and constructors, classes and interfaces, postcondition and precondition with respect to inheritance, old value collection and more.

1.5.2 Building a Tool

Besides the theoretical discussion we also provided a software tool, Jose, which demonstrates all those ideas using a real aspect oriented language, AspectJ. We see great importance to software tools in the adoption process of new methodologies. The abstract ideas and theoretical guidelines need tools and IDEs to reach the programmer. We hope Jose would help to ‘spread the word’ of design by contract.

1.5.3 DbC Variations and Extensions

The theoretical discussion and tool are used as a platform for extensions of the methodology as presented in Eiffel and on other Java DbC support tools. Such extensions include introducing a new concept of static invariant and support of reflexive contract enforcement (meaning enforcing the contracts of the methods appearing in the contract assertions). In addition we introduce here some alternative flavors of DbC which due to the tool are very easy to demonstrate. Such variations include a caller side contract - contract that is enforced not by the actual method executed but by the signature being called. Another issue is what we call implicit implication, which means to check the postcondition only under the condition of fulfilling the precondition.

1.5.4 AspectJ Semantics

The current semantics of AspectJ has some unintuitive aspects. We present a number of alternative semantics, and compare their expressive power. The “right” semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in practice. We hope that this paper will start a fruitful and constructive discussion on this question.

1.6 Related work

AspectJ’s Semantics

Many papers and books have been written about Aspect-Oriented Programming in general, and about AspectJ in particular (e.g., [6, 7, 12]), as well as several papers giving formal semantics of simple aspect-oriented languages (e.g., [9, 14, 20, 30, 32, 33]), but none of them provides a precise
(even if not completely formal) semantics of AspectJ. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ.

**DbC and AOP**

There are several discussions about the relation between DbC and AOP. Some of them claim that it can be done in AspectJ and give some examples [5, 13] others try to support their claims in formal semantics [18]. There is even a patent on this idea, submitted by some of the members of the AspectJ team [17].

**Other Tools**

There are a few ‘classic’ DbC tools for Java [4, 10, 16, 19, 23, 25]. By ‘classic’ we mean that the tool is responsible to instrument the given code and to embed the design by contract assertions in the right place. We know only about one tool — Barter [1] — that generates AspectJ’s code. Barter does not handle any of the difficult problems of design by contract. Its todo list includes: prevs, precondition and postcondition with respect to inheritance and more.

In the field of code generation tools there are some other tools that use AspectJ, as Cricket Cage [31] that automatically creates JUnit test cases, SuperJ [26] that transforms ‘classic’ superimpositions into aspects and a variant of Together for generating aspect code from UML models [8].
Chapter 2

Implementing DbC with AOP

2.1 Motivation

Consider the void method $m()$ of class $C$ with the precondition $alpha$. Naïve AspectJ code that enforces the contract for this method looks like:

```java
before() : call (void C.m())
{
    if (!alpha)
        throw new PreconditionViolationException();
}
```

Even this simple snippet raises implementation problems concerning AOP in general and AspectJ in particular.

**call vs. execution**

Using the `call` construct raises the following problem. A precondition is the client part of the contract, so it seems logical to use the `call` construct to capture this responsibility. That leads to two problems. First, due to AspectJ’s limitation all the source code should be available at compilation phase, which is not the case when calling advised methods from within a class whose source code we don’t have. For example, when using JUnit to call methods of the class, actually any use of the `callback method` scheme.

Another problem that occurs when using the `call` construct concerns potential polymorphism of $C$, the class in which $m()$ is defined. The type of the class to which the method belongs is determined by the `static` type of the `target` object at the call context. This behavior sometimes matches calls we didn’t intend to match and on the other hand skips desired calls. That problem conducted the study described in the appendix.

As a result of these problems we are using `execution pointcuts` instead of `call pointcuts`. Implications of using the `call` pointcuts are discussed later (Section 3.2).
The Context of the Contract

Another problem raised by the snippet above is the use of \texttt{alpha} as is inside an AspectJ’s advice. Although \texttt{alpha} stands for some boolean expression, this expression may contain elements known only in the context of the executing class \(C\), such as members of the class. Moreover, the expression may be significant only inside the body of the method, e.g. if it references parameters of the method. Inside the body of an advice the context is of the \textit{enclosing aspect}, so this wouldn’t work.

One solution for this problem is to \textit{understand} \texttt{alpha}. “Understanding \texttt{alpha}” means to \textit{parse} the expression string and to figure out the role of every subexpression in it. After doing so we can expose the context of the joinpoint using the \texttt{this()} and \texttt{args()} constructs\footnote{\texttt{this()} returns a reference to \texttt{this} and \texttt{args()} returns an array which contains the method’s arguments. See more at [28].} and bind them to some local arguments. Wherever there is a reference to some member of the class in the parsed string we could generate a method call or a field access using the exposed current object (which we bind to some local parameter using \texttt{this()} construct).

For example, suppose that \texttt{alpha} is ”\texttt{n}()>0” where \texttt{n}() in some member method of \(C\). The advice for that case is:

\begin{verbatim}
before(C that) : execution(void C.m()) && this(that)
{
    if (!(that.n()>0))
        throw new PreconditionViolationException();
}
\end{verbatim}

The major problem is not to \textit{instrument} \texttt{alpha} but is to \textit{understand} \texttt{alpha}. In the general case we need to be able to lexically analyze and parse every well formed Java boolean expression (using precedence rules, coercions and string literal inside the expressions). This can be done using a standard Java parser (such as ANTLR [29] with Java grammar), but here we prefer a more elegant solution.

Using Intertype Method Declaration

The solution for this problem is to use method \textit{intertype declaration} [28]. This mechanism (described in the AspectJ overview above - Section 1.3.2) allows to add (introduce) new methods to an existing class. The context of the introduced method is that of the executing object, exactly as if the method was written inside the class definition. For each assertion we introduce a corresponding method. The body of this method is a single \texttt{return} statement that returns the assertion. Inside the body of the \textit{advice} we just call the introduced method instead of checking the assertions directly. This idea takes care of all the context issues. In our example:

\begin{verbatim}
boolean C.check_precondition_m()
{
    return alpha;
}
\end{verbatim}
before(C that): execution (void C.m()) && this(that) {
    if(!that.check_precondition_m())
        throw new PreconditionViolationException();
}

Inheritance

Another issue (see Figure 2.1) that arises from this example is much more profound than all the above. It concerns the preconditions with respect to inheritance. Consider the m() method whose precondition we enforce. What if this m() was defined in some ancestor of C and was overridden by C? Think about the following situation: B is some class which defines a method m() with precondition beta. Class C extends B, and overrides m() with precondition alpha.

![Diagram showing inheritance and preconditions]

Figure 2.1: precondition enforcement with inheritance.

According to subcontracting rules, a client of C.m() should fulfill alpha || beta. What if she fulfills only beta but not alpha? In our example this scenario raises an exception while in fact there is no problem with it (refer to Section 1.2.2). The problem lies in our inability to collect the preconditions of all the levels in the hierarchy, and to throw the exception only if none of them is true. The following section presents some alternative designs that address this problem.

2.2 Preconditions

Let us consider several design alternatives to address the problem of subcontracting. Namely, introduce a mechanism to collect the preconditions up the hierarchy and throw an exception only if none of them holds.
Alternative I

While parsing the code we already know the hierarchy of the inspected system, therefore we could build (at compile time) the effective precondition (and later the effective postcondition) of each method. Effective precondition means that we can compose for every method the contract already ORed (or ANDed in case of postcondition). At runtime we will perform a single check per method by the actual type of its class.

The runtime performance of such an implementation is better, because during execution there is only single method call for each method execution. A potential problem could be when we would like to work with classes whose source code we don’t have. We might want to be able to have a jar file that contains AspectJ classes (aspects) that enforces the contract of some already compiled classes. In the current version of AspectJ the weaver must have the reference source code of the class it is manipulating.

Alternative II

The pointcuts are expressed in terms of execution joinpoints and within() constructs, hence the only matching joinpoint is the execution of the inspected method of the desired class (as in the previous alternative). The intertype method declared for each level checks the current level assertion and then calls the corresponding intertype method of the ancestor of the inspected class.

```java
01 boolean C.check_precondition_m()
02 {
03     return alpha || super.check_precondition_m();
04 }
05
06 before(C that) : execution (void C.m())
07      && within(C)
08      && this(that)
09 {
10     if (!that.check_precondition_m())
11         throw new PreconditionViolationException();
12 }
```

Figure 2.2: Precondition Enforcement. Alternative II: The chosen alternative.

This version was chosen due to its simplicity. The order of execution is very easy to understand and to maintain.

Short-Circuit Logical Operators, Execution Order and Exceptions

We are using here a short-circuit OR (the || operator as opposed to the ‘regular’ |). As soon as one level of the assertions satisfies the precondition, we do not check the rest of the assertions. In addition, if a method has more than one precondition assertions (in the same class) they are
all \textit{AND}ed, so any assertion violation in a specific level indicates a failure of the whole level (still not precondition violation). This ANDing should also be short circuited, and with strict evaluation order.

This thing is not only ‘lazy’ but also essential, because further redundant computations might end with throwing an exception which falsely implies a precondition violation. This situation is demonstrated in Figure 2.3. The class $C$ with a method $m()$ that have a precondition assertions $x!=null$ and $x.f()$. Calling the method $m()$ on an object of class $C$ when $x==null$ raises a null pointer exception if a short circuit \textit{and} is not used, and if the evaluation order is wrong.

This example reveals the issue of enforcement of \textit{execution order}. We ‘protect’ the potentially dangerous dereferencing by determining top down enforcement using the short circuit \textit{and}. The assertion enforced here is: $x!=null$ \textit{and then} $x.f()$. More about reflexive contracts (contracts of contracts) can be found in Section 3.3.

\begin{verbatim}
C
  m() @pre: x!=null
  @pre: x.f()
\end{verbatim}

Figure 2.3: The importance of short circuit precondition mechanism in preventing unwanted exceptions. The precondition $x.f()$ is ‘protected’ by $x!=null$ to prevent null pointer exception.

\section*{2.3 Postconditions}

The above discussion on preconditions is irrelevant to postconditions. Contrary to the precondition disjunction, postconditions are \textit{ANDed}. So every assertion violation indicates a postcondition violation, regardless of the class hierarchy. Hence our naïve enforcement do just fine. Let’s augment our example with postconditions:

\begin{verbatim}
B
  m() @pre: beta
  @post: delta
\end{verbatim}

\begin{verbatim}
C
  m() @pre: alpha
  @post: gamma
\end{verbatim}

Figure 2.4: Inheritance with postconditions.
There are several alternatives to implement postcondition enforcement. We present here two of them.

**Alternative I**

A naïve enforcement of the postcondition of method $m()$ of class $C$ is depicted in Figure 2.5.

```java
01 boolean C.check_postcondition_C_m()
02 {
03    return gamma;
04 }
05
06 after(C that) : execution (void C.m()) && this(that)
07 {
08    if(!that.check_postcondition_C_m())
09       throw new PostconditionViolationException();
10 }
```

Figure 2.5: Postcondition Enforcement: Alternative I.

The code which is generated to enforce $B$’s postcondition is very similar: $C$ is replaced with $B$ and the return expression in the intertype method is replaced with the right one ($\delta$). Since the pointcut matches calling $m()$ on the specified class and all of its descendants \(^2\), executing the method $m()$ of class $C$ triggers both pointcuts that call their corresponding $check\_postcondition\_x\_m()$s ($x$ stands for $C$ or $B$).

The intertype methods that enforce the postcondition have a unique name composed both of method signature and class name. This enables an advice to call an intertype method even if this method is not the lowest in the hierarchy (a leaf). In the precondition enforcement advice only calls a leaf intertype methods.

The methods are called with no determined order, since they were triggered by different advice of different aspects with no precedence declared. Since in that case the execution order does not matter, it is not enforced (but it can easily be done using the declare precedence construct).

**Alternative II**

Although precondition and postcondition differ in the logical operation between the hierarchy levels, they have much in common. The different treatment includes maintaining different sets of pointcuts and code duplication when writing the intertype method to check the assertion. The code snippet in Figure 2.6 is very similar to Alternative II of the previous section ($\gamma$ instead of $\alpha$ and $\text{pre}$ instead of $\text{post}$).

The operators $\&\&$ and $||$ stop the evaluation of the expression as soon as the result is determinable. Writing the intertype method assertion checker ($\text{check\_postcondition\_m}$ and

---

\(^2\) *execution* pointcuts include the descendants of the specified class implicitly. For further details see [2].
check_precondition_m) can be generalized using some AssertionChecker class with the descendants PreconditionChecker and PostconditionChecker that implement differently the methods assertionName() and logicalOperation().

```java
01 boolean C.check_postcondition_m()
02 {
03     return = gamma && super.check_postcondition_m();
04 }
05
06 after(C that) : execution (void C.m())
07     && within(C)
08     && this(that)
09 {
10     if(!that.check_postcondition_m())
11         throw new PostconditionViolationException();
12 }
```

Figure 2.6: Postcondition Enforcement: Alternative II.

### 2.4 Prev

The prev feature (keyword: $prev^3$) is used in postconditions to express the behavior of a method using a relation between expressions before and after the method execution. For example, adding an element to a set might include the postcondition: count() == $prev(count()) + 1. $prev(count()) returns the value as it was before the method execution whereas count() returns its value after the method execution. In Eiffel the keyword old is used instead of $prev [21].

It is clear that prev values should be stored somehow, and be passed to the contract enforcement method. Let us start with a naïve implementation of that feature, explain its pitfalls, and present solution alternatives. Consider the method m() of class C with the postcondition gamma. Suppose that gamma uses the original value of an expression gamma1 (i.e. gamma is: ...$prev(gamma1)...). Our code generator extracts the expression $prev(gamma1) out of gamma, stores it inside prev_gamma1 and then replaces $prev(gamma1) with prev_gamma1 inside gamma. We denote the modified gamma by gamma' (i.e. gamma' is: ...prev_gamma1...). Figure 2.7 shows the code generated to enforce the postcondition gamma.

Note that we changed the after advice to be an around advice, so we can easily pass data from before the execution of the method to the after advice. However, This code doesn’t compile – its first problem is that we don’t know the type of prev_gamma1 therefore we can’t define a local variable of that type, or declare properly the signature of check_postcondition_m (lines 01 and 10). This is not just a technical problem. Jose does not ‘know’ Java and hence could not deduce the type of the expression. (In fact, the real problem is even bigger; we can’t even

---

3Keywords in Jose start with $.
01 boolean C.check_postcondition_m(prev_gamma1)
02 {
03     return gamma' && super.check_postcondition_m();
04 }
05
06 void around(C that) : execution (void C.m())
07     && within(C)
08     && this(that)
09 {
10     prev_gamma1 = gamma1;
11     proceed();
12     if (!that.check_postcondition_m(prev_gamma1))
13         throw new PostconditionViolationException();
14 }

Figure 2.7: Postcondition with prev – a naïve implementation.

extract gamma1 from gamma. In the general case, gamma1 could be any legitimate Java expression and gamma could be any legitimate boolean Java expression, and therefore requires some parsing work).

This problem has two alternative solutions. 1) Ask the programmer to help at that point, and change the syntax of $prev(<expr>)$ construct to be $prev(<type> ; <expr>)$. 2) A more general solution is to use some standard Java parsing tool which already 'knows' Java grammar rules. Such a tool parses gamma1 and builds the corresponding AST. Given that tree we could compute the type of its root, using the information we have about the inspected class members and hierarchy structure.

As before, a greater challenge is caused by inheritance issues. Consider the ancestor class B, with the same method m() and a postcondition delta which includes a prev expression delta1. Before the execution of the method m() of class C we need to store in advance the prev value delta1 as well as gamma1. In the general case a method need to store all the prev values of all its ancestors.

We present two alternatives to solve the above problem:

**Alternative I**

To relieve each class of the obligation to know all its ancestors, we use the following scheme. We maintain a class hierarchy of prevCollectors corresponding to the hierarchy of the inspected classes. In the constructor of each of those collectors we store all the prevs needed by the assertions of the corresponding method in the specific class. Due to the mechanism of object construction in Java, whenever a constructor of some prevCollector is called, all of its super constructors are executed automatically, so all the prevs needed by the assertions of higher levels are handled automatically. All postcondition assertions now (such as check_postcondition_m in our example) have the same additional argument of prevCollector which includes all the stored
values needed for the assertions along the hierarchy.

Though the scheme is beautiful, it has some faults. First, it increases the number of classes created by our tool. Without this feature we create one aspect per each inspected class, and now we create also one additional class for each method. Besides, it is not simple enough: there are many subtle things to care of, such as different treatment for interfaces, extensive use of casting, and delicate treatment in levels with no prevs at all (that still need to support the prevCollector mechanism to make it work for their descendants). We need to look for simpler alternatives.

**Alternative II**

We distinguish the precondition and postcondition checking process. We have two kinds of pointcuts, the original one for precondition which matches a specific method and handle the hierarchy through super calls in the inter type assertion checker methods (described in Section 2.2, Alternative II). The new family of pointcuts matches a method signature execution along the hierarchy, this pointcut is advised by an around advice which first collects the prev values if needed and then calls the intertype postcondition checker method which checks the assertion and does not call its super. Because of the pointcut definition the super assertion will be called through its ‘own’ advice (we do not use within in the pointcut definition).

This alternative is depicted in Figure 2.8.

```java
01 boolean C.check_postcondition_C_m(type_gamma1 prev_gamma1)
02 {
03   return gamma;
04 }
05
06 void around() : execution (void C.m()) && this(that)
07 {
08   type_gamma1 prev_gamma1 = gamma1;
09   proceed();
10   if (!that.check_postcondition_C_m(prev_gamma1))
11     throw new PostconditionViolationException();
12 }
```

Figure 2.8: Postcondition with prev – alternative II.

Note that now we don’t use the within construct in the pointcut definition, so any method call for m() on object of type C, or one of its descendants matches the pointcut, and therefore triggers the advice. This alternative is the one implemented in Jose.

There remains the issue of precedence. It is easy to enforce precedence using the declare precedence construct, although it is not necessary here. All $prev$ expressions should not have any side effects, and – opposed to the precondition case – there could not be a situation where an exception is thrown although the postcondition already holds.

Enforcing the bottom-up order uses the scheme of declaring the lower class to have higher precedence of its ancestors. This is done in pairs, we declare the corresponding aspect of each
class as having precedence over the corresponding aspect of its super class. In our example:

declare precedence: C_correspondingAspect,
                  B_correspondingAspect

2.5 Class Invariants

```
B @inv: sigma

m()


C @inv: omega

m()
```

Figure 2.9: Inheritance with invariants.

Class invariants are boolean assertions that express a property of the class. Those assertions are enforced after the constructor is executed and before and after every non-public instance method execution. Upon the entry to the constructor the object is not yet created, and private methods are not part of the class contract.

After putting so much effort in solving the delicate matters in precondition and postconditions, dealing with invariants is fairly easy. Invariants are easier to implement since they do not have the difficulty of preconditions that need to be collected nor the difficulty of postconditions that need their prev to be stored in advance.

We decided to use the precondition mechanism to handle also invariants, although other alternatives could be implemented as well.

The code in Figure 2.10 is generated for the class C depicted in Figure 2.9.

We prefer to use before and after advice rather than the around advice due to its performance overhead. Note that the check_invariant method is being created only once per class but every (non-private instance) method triggers it using wildcards. Invariant checking before and after constructor execution are discussed in the next section.
01 boolean C.check_invariant()
02 {
03     return omega && super.check_invariant();
04 }
05
06 before(C that) : execution(!private !static * C.*(..))
07     && within(C)
08     && this(that)
09 {
10     if(!that.check_invariant())
11         throw new InvariantViolationException();
12 }
13
14 after(C that) : execution(!private !static * C.*(..))
15     && within(C)
16     && this(that)
17 {
18     if (!that.check_invariant())
19         throw new InvariantViolationException();
20 }

Figure 2.10: Invariant enforcement.

Figure 2.11: Inheritance with constructors.

2.6 Constructors

In Java, constructors are not inherited. When a constructor calls its super constructor, it is a normal call, which must obey the callee’s precondition. Therefore, preconditions should be
checked separately for each constructor, as it is called.

The situation for invariants is different. Invariants of all superclasses must be satisfied when the new object is ready, i.e. at the end of the first constructor (the lowest in the hierarchy). It is wrong to check invariants when super constructors finish execution, since the object is not yet fully initialized. This applies even to inherited invariants. For example, an abstract Stack class might define the invariant capacity() >= 0. In this class, capacity() is a getter method for an internal field, which is initialized to 0. However, the class ArrayedStack, which extends Stack, may redefine capacity() to return rep.length, where rep is an array. The constructor for ArrayedStack creates this array, but it is still null when the super constructor (from Stack) finishes. If the invariant (from Stack) is evaluated at this point, a NullPointerException will be generated. Therefore, it is necessary to evaluate all invariants, including inherited ones, only at the end of the first constructor.

Constructor postconditions pose the greatest difficulty. For the reasons explained above, they cannot be evaluated at the end of the super constructors (think of the postcondition capacity() == n for the constructor of Stack, where n is the constructor’s parameter). But they cannot be evaluated at any later point, since there is no requirement that they hold later. Therefore, only the postconditions on the first (the ‘lowest’) constructor should be evaluated at all. This is consistent with the theory, because postconditions need to be repeated for ArrayedStack anyway, as there is no inheritance relationship between constructors.

As constructor behavior differs from a regular method behavior we can’t use the same scheme we used for methods. With method calls we checked the super class’s contract although the super method actually was not called, as opposed to the constructor case, where super constructor must execute before the execution of a lower lever constructor.

### Enforcing the Constructor Contracts

The only thing we need in order to enforce the constructor contract is a way of identifying the first constructor, as distinct from super constructors. This will do the trick both for invariants and for postconditions.

Preconditions should be checked for every constructor, without inheritance, and before the super constructor. Execution join points happen after the call to the super constructor, so we need to use a preinitialization pointcut instead of execution. (Preinitialization join points cover the code executed between the start of the constructor execution and the super call; this code includes the computation of the parameters to the super constructor.)

Following (Figure 2.12) is the code generated for the constructor of class C in Figure 2.11. Constructor arguments and prevs are handled the same way as before so we ignore them for now with no lose of generality.

This will capture all executions of the constructor of C that is not a super constructor, because in a super constructor the actual class of the current object will be a subclass of C. The after advice uses its pointcut to put the checks of all the invariants, and only C’s postcondition. Both for invariants and postconditions only the lowest constructor is matched but only the invariant intertype method assertion includes a super call. Note that the check invariant method is being created only once per class but every method execution triggers it.
boolean C.check_precondition_C_new()
{
    return tau;
}

boolean C.check_postcondition_C_new()
{
    return rho;
}

// enforcing constructor precondition
before(C that) : preinitialization (void C.new())
    && within(C)
    && this(that)
{
    if(!that.check_precondition_C_new())
        throw new PreconditionViolationException();
}

// enforcing invariant upon the end of constructor
after(C that) : execution (void C.new())
    && this(that)
    && if(that.getClass() == C.class)
{
    if (!that.check_invariant())
        throw new InvariantViolationException();
}

// enforcing constructor postcondition
after(C that) : execution (void C.new())
    && this(that)
    && if(that.getClass() == C.class)
{
    if (!that.check_postcondition_C_new())
        throw new PostconditionViolationException();
}

Figure 2.12: Enforcing constructors contracts.

Prev Collection with Constructors

1. The postcondition of super constructor is not checked after performing the constructor
   of a descendant class, therefore one should not take care of storing the prevs of such
postconditions. That conforms with the prev storage scheme presented earlier.

2. During the development of Jose we declared a designated field in the aspect to store the prev value. To support canonical and recursive calls we declared the aspect to be perthis, i.e. we instantiate a new aspect to handle any new inspected object. This scheme doesn’t work on the constructor case; as before the constructor there is no object yet to relate to. We solve this problem by storing the prev value as a local variable of the advice.

### 2.7 Interfaces

The differences between interface and ‘regular’ class are as followed. First, an interface has only method signatures with no bodies, therefore a pointcut which include the construct within with some interface name would never match the execution of any method. It is clear that an interface could never be the dynamic type of the current object.

Second, an interface can not be accessed through the super construct. The super construct refers only to the class which was extended and not the implemented interface.

The above differences imply that interfaces should be handled explicitly. The execution pointcut which serves the precondition enforcement mechanism should not be generated (it is only used by the lowest class in the hierarchy), the postcondition mechanism may stay as is, and the introduced assertions should explicitly call all their interface’s corresponding assertions (not via super calls). Introduced assertions of interface’s method should also call their implemented interface’s assertions to support interface hierarchy.

The following code (Figure 2.14) is generated for invariant enforcement of the interfaces at Figure 2.13 (precondition and postcondition enforcement mechanism follow the same convention exactly and are not shown here).

In the general case, a class or an interface may implement more than a single interface. In that case for every class or interface we generate an intertype method (check_invariant) which checks the precondition of that level and explicitly calls the intertype methods declared for only its direct implemented interface (check_invariant_xx). The recursive nature of that scheme takes care of all the ancestors along the hierarchy tree.

In Figure 2.15 class C extends class B and implements two interfaces – I1_1 and I1_2. The interface I1_1 also implements two interfaces – I2_1 and I2_2. Each of the intertype methods declared for each interface should explicitly call only the corresponding method in the implemented interfaces.

Following the same rule, class C should explicitly call its super’s corresponding method and the methods of its direct implemented interfaces, I1_1 and I1_2:

```
1 boolean C.check_invariant()
2 {
3     return psi && super.check_invariant()
4         && check_invariant_I1_1()
5             && check_invariant_I1_2()
6             ;
7 }
```
boolean I2.check_invariant_I2()
{
    return omega;
}

boolean I1.check_invariant_I1()
{
    return xi && check_invariant_I2();
}

boolean C.check_invariant()
{
    return psi && super.check_invariant() && check_invariant_I1();
}

Figure 2.13: Class hierarchy with interface implementation.

Figure 2.14: Enforcing class invariants with interfaces.
Figure 2.15: Class hierarchy with multiple interface implementation.

```java
01 boolean I2_1.check_invariant_I2_1()
02 {
03     return omega ;
04 }
05
06 boolean I2_2.check_invariant_I2_2()
07 {
08     return omega';
09 }
10
11 boolean I1_2.check_invariant_I1_2()
12 {
13     return xi';
14 }
15
16 boolean I1_1.check_invariant_I1_1()
17 {
18     return xi && check_invariant_I2_1()
19         && check_invariant_I2_2()
20         ;
21 }
```
Chapter 3

Variants and Extensions of DbC

While working on Jose, we had some references to relate. The DbC approach, first presented by Bertrand Meyer and implemented in Eiffel [21] and there exist a family of Java DbC tools such as Jass [4], JML [16], iContract [10, 25], JContract [23] and JMSAssert [19]. During the development phase we tried to conform to JMSAssert syntax and features.

After establishing a working configuration with the desired basic feature, we encountered some immediate extension to the DbC methodology that do not appear in the referenced tools. These extensions are very easy to implement, and the Jose platform made it very easy to test and to experiment with.

3.1 Class Oriented Contracts

Design by Contract assertions are object oriented by nature (opposed to class oriented). A technical implication of it is the fact that invariants are not being checked before a constructor or around static methods. It raises the question, are there cases where an invariant check should occur especially on those points.

There are some class properties that can be expressed using the existing constructs. Consider a final class that manages the number of its instances using a static class member numOfInstances. Each time a new instance is created this counter is incremented. We are considering a final class to avoid inheritance issues. The class also wants to limit the number of instances to be below some limit. The invariant of such class probably includes the assertion:

\[ \text{numOfInstances} < \text{limit} \]

And upon the exit from the constructor we encounter an invariant violation. This is too late. An invariant violation suggests that the fault is of the class, where here it seems to be the fault of the client, the one that invoked the constructor. To fix that we add a precondition for the constructor:

\[ \text{numOfInstances} < \text{limit} - 1 \]
This time as soon as a careless client calls the redundant constructor, a precondition violation occurs, which puts the blame on the right side. We presented here a case where a class oriented assertion was enforced by the existing mechanism of DbC although it is object oriented. It is not always the case as depicted in the following case.

Consider a class $C$ with some property $p$ (a class invariant) which does not include any reference to non-static members of $C$. Such $p$ may express a consistency between some static fields of the class, e.g. $\text{UpperLimit} > \text{LowerLimit}$. This property is checked upon the entry and exit of each of $C$'s non-static methods. What if $C$ has a static method $f$ which violates the property $p$? In the current mechanism of DbC this violation is not be intercepted on time, but only upon the invocation of the next non-static method of $C$.

One could ask about the nature of this violation. Such a violation suggests a bug in the implementation of $f$ exactly as the ‘regular’ invariant should expose. Adding precondition and postcondition for every static method also solves this problem, but this is exactly what tells us that this property should be managed centrally (this is an aspect of the class). Note that ‘regular’ invariant can also be replaced by pre- and postcondition for every method.

We suggest to add a new construct to the DbC language. @static_inv assertions are enforced upon every entry and exit of every method or constructor of the class. It is clear that these assertions may only include static references and operations.

### 3.2 The Client Perspective

#### 3.2.1 Dynamic vs. Static Contracts

What if the client satisfies in the contract by mistake due to the dynamically dispatched method?

Recall Figure 1.1. $B$ inherits from $A$ and redeclares $r()$. Consider the plight of the client. In the call to the method $r()$ the target may, out of polymorphism, be of type $B$ rather than $A$. But $C$ does not know about this! In fact $C$ may use $B$ without its author ever knowing about the existence of such class.

Let us discuss the precondition enforcement mechanism on such scenario. The joinpoint that triggers the precondition enforcement assertions is of type execution joinpoint with conjunction of a within() construct. That means that the assertions being executed are of the actual method which was invoked. The client doesn’t know that behind the scenes a precondition was not aware of may be enforced.

As we mentioned before, a precondition may only be weakened in descendant classes, so if the precondition holds for some ancestor class it should be kept true for all its descendants. The problem comes from the other side - aren’t we too conservative here? Aren’t we giving the client a credit she might exploit?

Consider a client of type $C$ that calls the method $r()$ without fulfilling its precondition $\alpha$ (Figure 1.1). The static type of the target object is $A$ so this should indicate a precondition violation, a client fault (bug). But what if the client does fulfill $\gamma$? Because of our precondition enforcement mechanism it is enough to fulfill $\alpha \parallel \gamma$ for the precondition to hold so the precondition here holds “by accident”. If our client (the class $C$) had a reference of type $B$ there wouldn’t be any problem. In such case the client knows that it references an object of type $B$.
(or one of its descendants), and therefore it should fulfill \texttt{gamma} which it does.

The problem exists when a client does not satisfy the precondition of the \textit{static type} which it references. In some cases the client might end in fulfilling the precondition because of the dynamic type that weakens the precondition - how can we avoid such cases? Do we want to prevent such scenarios?

When we called our enforcement mechanism \textit{conservative}, we imply that no \textit{harm} can be done. Harm here means that the execution of some method may encounter a situation which it didn’t expect to due to its precondition. This is impossible here. A method can not be executed unless its precondition is satisfied. Nevertheless, this situation indicates a design problem - if there are some assumptions that the client writer takes into account(e.g., she knows in advance the actual type of the supplier class would be a descendant one), they should be explicit in the code. On the other hand if \textit{all} the descendants of some class weaken a certain precondition this precondition should be moved up in hierarchy to the common ancestor.

The situation here resembles the explicit casting required by Java. That happens when an object of some type is assigned to a reference of different type. Even if this assignment is “safe”, it ends with a compilation warning. This requires the client to be explicit in potentially dangerous operations.

How can the client be explicit in the scenario above? She could have checked the actual type of the object it addresses (where \texttt{a1} is a reference of static type \texttt{A}):

\begin{verbatim}
if (a1 instanceof B)
   a1.r();
\end{verbatim}

Although it is explicit, this is a wrong object oriented programming style. If \texttt{r()} is a polymorphic operation, it should be used the same way for all types. It can be a bug in the design of the client side, who should handle the same way all descendants of \texttt{A}. It can also indicate a wrong design on the server side (hierarchy of \texttt{A}). If the precondition \texttt{gamma} is common to all the descendants of \texttt{A}, it should appear in their common ancestor (be the precondition of \texttt{A} itself).

3.2.2 Taking Client’s Perspective in Jose

How can we implement the “client side contract enforcement” in AspectJ? We would like to see things from the caller point of view, and to check only the preconditions derived from the static type of the references.

This is exactly the difference between the \textit{execution} joinpoint we used and the \textit{call} joinpoint. When specifying a class name inside the definition of a call pointcut, this type is considered as the static type of the target object. For more details about that refer to the appendix that deals with call and execution semantics in AspectJ.

Nevertheless, it is not so simple. The greatest difficulty here is to collect the preconditions. We need to collect them from all levels of the hierarchy that are ancestors of the static type of reference used in the call (we are going to show later how is it done). Note that a precondition violation happens on if all precondition assertions of \textit{all} levels fail, so we need a mechanism that first checks all the different levels and then decides if to generate a precondition violation or not.
We demonstrate the difficulty using the example in Figure 3.1. As before, we generate an

![Diagram showing precondition enforcement with inheritance.]

Figure 3.1: precondition enforcement with inheritance.

intertype method to enforce the precondition of every class.

```java
boolean A.check_precondition_r()
{
    return alpha;
}

boolean B.check_precondition_r()
{
    return beta || super.check_precondition_r();
}
```

The problem arises at the pointcut definition. This time we want to take the caller perspective, so we use a call joinpoint instead of an execution joinpoint. A naïve rewriting of the pointcuts from Section 2.2 helps to emphasize the difficulties in the implementation of such scheme:

```java
01 before(A that) : call (void A.r()) && target(that)
02 {
03     if (!that.check_precondition_r())
04         throw new PreconditionViolationException();
05 }
06
07 before(B that) : call (void B.r()) && target(that)
08 {
09     if (!that.check_precondition_r())
10         throw new PreconditionViolationException();
11 }
```
Beside the use of `call` instead of `execution`, there is an additional important difference. The `within` construct has different meaning in this context (it means that call to a specific method can be done only by the same class that declares this method). At Section 2.2 it was used to distinguish between the different versions of the same method (overridden versions), but in this case it is the other way around: the client need not to be aware of the actual type of our server.

The lack of the `within` creates an unexpected problem. We demonstrate it using the following snippet:

```java
A a = new A();
B b = new B();
A ab = new B();
```

The references `a` and `b`, each has the same static and dynamic type, while `ab` is a polymorphic reference: its dynamic type is subtype of its static type. When we call `a.r()`, only the first pointcut matches the call, and the “right” `check_precondition_r` is executed.

Calling `ab.r()` raises the first problem. The call is matched only by the first pointcut (which is just what we wanted). The object which `ab` is referencing has two `r()` methods – declared by `A` and `B` – and two `check_precondition_r` methods, declared by their corresponding aspects. Exactly as `B`’s `r()` overrides `A`’s `r()`, so does `check_precondition_r` which declared by `B`’s corresponding aspect overrides `A`’s `()` corresponding aspect `check_precondition_r`. So when the before advice calls the intertype declaration method to check the precondition - it ends with calling the wrong method. Although the method dispatch is made dynamically, we want to enforce the contract based on the information available to the client.

This problem can be easily solved by giving each intertype method a unique name that contains also the name of the class in which it was put. For example `check_precondition_r` becomes `check_precondition_A_r` when introduced to `A` and `check_precondition_B_r` when introduced to `B`. This was not done in order to maintain for each inspected method hierarchy a corresponding enforcement method hierarchy.

A greater problem arises when calling `b.r()`. This time both pointcuts match this call, and the precondition enforcement is executed twice. After fixing our previous bug – so every pointcut can call the assertion method which was defined by its enclosing aspect – we get different checks in the different pointcuts which is a serious mistake.

Consider a client that keeps `beta` but violates `alpha`. The call `b.r()` triggers `B`’s aspect’s pointcut which ends with precondition satisfaction but also `A`’s aspect’s pointcut which ends with precondition violation - which is not the case here.

### 3.2.3 As Good as it Gets

Under the current semantics of AspectJ it is tricky to implement the “client side contract” described above. The type which appears in the `call` pointcut definition implicitly includes all its subtypes. All the subtypes should be explicitly eliminated in the pointcut definition. As there is no corresponding construct to `within` for `call` and we can’t use `target` which is based on the `dynamic` type, so we need to add negated `call` pointcuts for each of the static type subclass.
Consider a class $A$ with $k$ descendants (Figure 3.2). At the pointcut definition we need to explicitly eliminate all its descendants:

01 before(A that) : call (void A.r())
02       && !(call (void B1.r()))
03       && !(call (void B2.r()))
04       ...
05       && !(call (void Bk.r()))
06       && target(that)
07 {
08       if(!that.check_precondition_A_r())
09           throw new PreconditionViolationException();
10     }

This is highly problematic if we intend to jar our classes along with their contracts, and expect a future extension of them by other classes. Those classes are not excluded in the already compiled contract.

Implementing the client perspective under different semantics\(^1\) \((\text{static-narrow})\) would yield a much more elegant solution then the above (no need to eliminate subtypes because nothing is implicitly included).

### 3.3 Recursive and Reflexive Assertions

#### 3.3.1 Eiffel Solution

In Eiffel no assertions are checked in the control flow of the evaluation of the top assertion. The Assertion Evaluation rule [21, p. 403] says that “During the process of evaluating an assertion

\(^1\)Alternative semantics are presented in [2, Section 3].
at run-time, routine calls shall be executed without any evaluation of the associated assertions”. Meyer explains:

If a call to \( f \) occurs as part of assertion checking for \( r \), that is too late to ask whether \( f \) satisfies its assertions. The proper time for such a question is when you decide to use \( f \) in the assertions applicable to \( r \).

Meyer uses the following analogy. Think of \( f \) as a security guard at the entrance of a nuclear plant, in charge of inspecting the credentials of visitors. There are requirements on guards too. But you will run the background check on a guard in advance; not while she is screening the day’s visitors.

By not checking, Meyer takes us to the other extreme, instead of checking our guard continuously we don’t check her at all. When is “in advance”? When does one decide to use a certain function in the assertions applicable to another?

Meyer concludes by saying “The assertion language can express many higher-level properties through function calls, although the functions involved must be simple and of unimpeachable correctness.” Many of the programmers we know (especially beginners) claim that the code they write is simple and unimpeachably correct and therefore should not be checked or asserted.

We object to the distinction made between methods written in the source code, and methods in the contracts. If we could be certain “in advance” about the use of \( f \), why can’t we be certain “in advance” on the use of \( r \) and spare the use of design by contract altogether? We claim that there are some cases where also the assertion’s contracts need to be enforced. The different cases are described in the following sections.

### 3.3.2 Infinite Recursion

There is one case in which we surely do not want to enforce an assertion’s contract. That is the case where the enforcement ends with an infinite recursion. Let’s discuss the different kinds of potential recursion that may occur when enforcing a contract.

1. The first case is an invariant that includes a method call. E.g. \( \text{INV count()>=0} \). An invariant should be checked upon any method invocation and method exit. Each time the assertion is executed, we need to evaluate the included method (in our example the \( \text{count} \) method), which is by itself a method invocation so it requires an invariant check before executing and we get an infinite recursion.

2. Another case of contract recursion can happen if there are \textit{mutual} calls from within different methods’ contract to one another (a method could also reference itself in its own assertions which leads directly to infinite recursion). For example: a method \( r \) can have a precondition \( f \) which during its execution calls \( r \). The same holds if \( f \) is the postcondition of \( r \). And of course that \textit{cyclic reference} can be even more complex.

3. A third family of potential recursion is when the assertion function of some method has the same method as its own precondition. For example: a method \( r \) can have a precondition \( f \) which has a precondition \( r \).

This raises some questions about the \textit{meaning} of the contract of the contract.
3.3.3 The Meaning of Assertions of Assertions

Let’s start with invariants. What is the meaning of checking the invariant while evaluating it? What are the implications of preventing this recursion?

A class invariant deals with the state of an object. The class invariant should be checked upon any method (class operation) entry and exit. This resolution reflects the state of the object as it seems to its clients; this is also the reason why private methods are not considered as checkpoints for invariant enforcement.

Checking the invariant while evaluating it is redundant (besides the technical difficulty of avoiding the infinite recursion). Given that all design by contract assertions should be non-destructive (non-destructive assertions would be dealt in Section 3.5), nothing could have happened to the state of the object during the evaluation of the invariant itself.

What about the other cases? A method may be used both in the source code and on another method’s pre- or postcondition (We ignore the case where a method has a contract that includes calls to itself – this is wrong programming, it is just as trying to save yourself from drowning by pulling your own hair...). Take the Stack class for example. The method top() is part of Stack’s interface, but it is also used inside the postcondition of push(Object x):

@pre: top()==x. top() also has a precondition: @post: !empty(). When we decide to enforce top’s precondition we implicitly say we don’t trust Stack’s clients to fulfill its precondition. We can’t make an exception for push. Consider a case where a method is especially written to be used inside some other method’s precondition. This method’s postcondition describes what this method does. The fact that its client is another method’s precondition should not affect checking its own postcondition.

One problem remains. When we allow the evaluation of assertions’ associated assertions we might find ourselves inside an infinite recursion. These cases we want to avoid.

3.3.4 Preventing Infinite Recursion in Jose

We prevent assertion evaluation only in the control flow (using cflowbelow()) of the intertype assertion checker (check_precondition_m). We do that by excluding these joinpoints in the pointcut definition (of both types: pre- and postcondition ones). This allows us to check the precondition of the method inside the precondition of another method as long as there is no cyclic reference.

Doing so for class C with method m() and precondition alpha yields the code in Figure 3.3. The same technique is also used for postcondition and invariants.

Note that the pointcut definition prevent a recursion of the method check_precondition_m() even if the inner method is acting on a different object than the outer method.

2Note, that a method call inside the invariant expression triggers an invariant check of its enclosing class.
01 boolean C.check_precondition_m()
02 {
03     return alpha || super.check_precondition_m();
04 }
05
06 before(C that) : execution (void C.m())
07     && this(that)
08     && within(C)
09     && !cflowbelow(execution(C.check_precondition_m()))
10 {
11     if (!that.check_precondition_m())
12         throw new PreconditionViolationException();
13 }

Figure 3.3: Preventing Infinite Recursion.

3.4 Precondition Implies the Postcondition Implicitly

3.4.1 Two Flavors of Postconditions

A contract defines the responsibilities between the client and the supplier. Recall that if the client
(the caller to the method) fulfils the method’s precondition, the supplier (the method writer)
guarantees the termination of the method and the postcondition upon method termination. If
the client does not satisfy the precondition at the entry point of the method, the supplier is free
of her obligations.

Consider a class $C$ with a method $m$ that has a precondition $\alpha$. If the method writer guarantees
that upon termination $\beta$ holds, we can simply say that the postcondition of this method is $\beta$. Furthermore, we could say that the method postcondition is $\alpha \rightarrow \beta$ (we call it the explicit
postcondition). By the truth table of the ‘$\rightarrow$’ operator, if $\alpha$ is false (the precondition doesn’t
hold) then the value of the expression is true (the supplier guarantees nothing). This way the
postcondition is expressed explicitly as a derivation of its precondition.

The contract between the client and the supplier can be formulated as “precondition implies
postcondition”. If the postcondition is implicit it is: $\alpha \rightarrow \beta$. If the postcondition is explicit it is
$\alpha \rightarrow (\alpha \rightarrow \beta)$.

Note that the two expression $\alpha \rightarrow \beta$ (precondition implies postcondition) and $\alpha \rightarrow (\alpha \rightarrow \beta)$
(precondition implies explicit postcondition) are equivalent. As long as there is no inheritance
involved, there is no reason to express each postcondition as implied by its precondition explicitly.

The postcondition is evaluated in the context of the method after executing it. $\alpha$ is an
expression that need to be evaluated before the execution of the method and might change
along the execution of $m$. Therefore the ‘real’ postcondition is $prev(\alpha) \rightarrow \beta$ and not $\alpha \rightarrow \beta$.

Following is a scenario where there is a difference between the two alternatives of the post-
condition formulation. As indicated before, the example involves inheritance. Consider the class
Set with the method $\text{insert}(\text{Object } x)$. The method gets as an argument an element that is
not already included in the set and adds it. The precondition of the method is !member(x) and its postcondition is:

\[ \text{count} == \$\text{prev(count)} + 1. \]

BetterSet is a descendant class of Set. It has an improved (‘better’) insert method. The new insert allows as arguments elements that are already included in the set. In such a case insert doesn’t do anything. Followed the subcontracting rules – BetterSet weakens its super class’s precondition and turns it to true. But what about the postcondition? For some inputs the new method contradicts the postcondition of the super class. This also contradicts our rule to only strengthen postcondition in descendant classes.

If we had used the explicit postcondition in the base class the problem would be solved. The postcondition in Set would be:

\[ \$\text{prev}(!\text{member(x)}) \rightarrow \text{count} == \$\text{prev(count)} + 1 \]

And the additional postcondition of BetterSet would be:

\[ \$\text{prev}(\text{member(x)}) \rightarrow \text{count} == \$\text{prev(count)} \]

Recall that we make a conjunction of the two when evaluating the postcondition of the descendant class.

3.4.2 Implementing Explicit Derivation in Jose

We want to give the contract writer a mechanism to write its postcondition implicitly, and to automatically enforce the explicit contract. For achieving that, we transform each postcondition \( \beta \) into an explicit postcondition. We use the logical equivalence of \( \neg\text{prev}(\alpha) \lor \beta \) and \( \text{prev}(\alpha) \rightarrow \beta \). Therefore we replace each \( \beta \) with \( \neg\text{prev}(\alpha) \lor \beta \).

For example, recall our class \( \mathcal{C} \) with method \( m() \), a precondition \( \alpha \), and a postcondition \( \beta \). If we are working on implicit postcondition mode, it generates the code in Figure 3.4.

Note that we didn’t use alpha inside the advice because alpha need to be evaluated in the context of the class. We also can’t use our ‘regular’ intertype method that enforces the precondition (check_precondition_m() returns the disjunction of the precondition along the hierarchy and we want only alpha). Therefore we create a new intertype method, check_precondition_current_Level_m(), that evaluates alpha with no super calls.

3.5 Non Destructive DbC Assertions

It is clear that DbC assertions must not change the state of the system they inspect. A state of an object is considered to be the value of all its fields. Unlike C++, Java does not use the const modifier to indicate that a certain method does not make any changes to the object fields. A careless contract writer can easily make foolish things such as changing the values of any of those fields.
void around(C that) : execution (void C.m()) && this(that) {
    boolean preconditionHeld = that.check_precondition_current_Level_m();
    proceed();
    if (!that.check_postcondition_C_m(preconditionHeld))
        throw new PostconditionViolationException();
}

boolean C.check_precondition_current_Level_m()
{
    return alpha ;
}

boolean C.check_postcondition_C_m(boolean preconditionHeld)
{
    return !preconditionHeld || beta ;
}

Figure 3.4: Implementing Explicit Derivation.

To know if a method is non-destructive with respect to some class we only need to verify that during the control flow of its execution it doesn’t make any assignment to any of the class members. This can be easily done using the \texttt{cflow} construct.

For every intertype method which was generated to enforce invariant, pre- or postcondition assertion, we declare a pointcut that matches all member assignments (set joinpoints) under the control flow of this method. Any matched joinpoint indicates a forbidden assignment. An exception is thrown to indicate the problem.

For example, recall class \texttt{C} with method \texttt{m()} a precondition \(\alpha\) and a postcondition \(\beta\). For the method \texttt{check_postcondition_C_m()} that enforces \texttt{m}'s postcondition we generate the advice in Figure 3.5.

before () : cflow(C.check_postcondition_C_m()) && set(* C.*)
{
    throw new DestructiveDBCAssertionException();
}

Figure 3.5: Detect destructive assertions.

The same technique also applies to \texttt{m}'s \textit{precondition} and to \texttt{C}'s \textit{invariant}.

\textbf{An Elaboration.} Some of this information can be known \textit{statically} (at compilation time), but due to the limitations of \textit{Jose} (at the current version, we don’t analyze the structure of
the assertions) we don’t know the name of the methods composing the boolean assertion. If we would, we would have used `declare error` construct to find `assignments` (set joinpoint) to class members within the code of those methods (using `within code` construct). Note that this does not replace the cflow pointcut shown above. Assignments may also appear within methods that were called by a top level method and wouldn’t be caught by the `within` pointcut.

Two problems remains:

1. The pointcut `set(* C.*)` matches only assignments for C’s fields. It does not detect changing objects of different classes. We can not fix it with the pointcut `set(* *)` because it matches too much (it matches also changing objects which are created by `check_postcondition C_m()`).

2. We only check *concrete* side effect rather than *abstract* side effect [21]. There are assignments that do not change the behavior of the object.
4.1 Other Languages and Methodologies

The ideas presented in *Jose* can be implemented with other aspect oriented languages. In fact, there are other programming languages that with their corresponding aspect mechanism can gain design by contract capabilities.

Moreover, implementing design by contract using AspectJ is just a case study in the context of *generative programming*. Software construction doesn’t end with design by contract. Some of the ideas can be applied over other methodologies to extend the underlying programming language with features it lacks (E.g. one could try to add the construct `const` to Java using generative programming in AspectJ).

4.2 AspectJ Semantics

There is no precise (even if not completely formal) semantics of AspectJ. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ.

The current semantics of AspectJ has some unintuitive aspects. We have presented a number of alternative semantics, and compared their expressive power. The “right” semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in practice. We hope that this paper will start a fruitful and constructive discussion on this question.

AspectJ acts not only in the context of other aspect oriented programming languages, but is also influenced by other methodologies in which one can find much similarity to the principles presented in aspect oriented programming. Some of them are Superimpositions [27], Slicing [15], Multi-Dimensional Separation of Concerns and Composition Filters [11]. Making a reduction from those areas to AOP may yield some benefits in formalizing the desired semantics of AspectJ.
Appendix A

Call and Execution Semantics in AspectJ

Authors: Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai.

Presented at Foundations of Aspect-Oriented Languages (FOAL) 2004. FOAL 2004 was held in conjunction with AOSD 2004 in Lancaster, UK, on Tuesday, 23 March 2004.
ABSTRACT
The Aspect-Oriented Programming methodology provides a means of encapsulation of crosscutting concerns in software. AspectJ is a general-purpose aspect-oriented programming language that extends Java. This paper investigates the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models.

Categories and Subject Descriptors
D.3.2 [Programming Languages]: Language Classifications—Object-oriented languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Operational semantics

General Terms
Languages

Keywords
Aspect-oriented programming, AspectJ

1. INTRODUCTION
Many papers and books have been written about Aspect-Oriented Programming (AOP) in general, and about AspectJ in particular (e.g., [1, 2, 4]), as well as several papers giving formal semantics of simple aspect-oriented languages (e.g., [3, 5, 6, 8–10]), but none of them provides a precise (even if not completely formal) semantics of AspectJ. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ. For example, we are developing a design-by-contract [7] tool for Java. The main purpose of such a tool is to instrument the code to check assertions (method pre- and postconditions and class invariants) at run time. Existing tools we have examined perform this instrumentation in various ways, all of which have subtle errors. Our tool uses AspectJ instead of ad-hoc methods. While working on the tool, we discovered that some pointcuts we wrote did not yield the sets of join points that we expected. This has led us to conduct the study that we report on here.

We believe that a close examination of the semantics of AspectJ as manifested by the current implementation, and a discussion of the desired or “correct” semantics, is important to the AOP community. We hope that studies of the semantics of other parts of the language will follow. This paper investigates one of the subtle parts of AspectJ, namely, call and execution pointcuts and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models. We note that Jagadeesan et al. [3] mention a few of these shortcomings, but do not discuss their deficiencies.

We follow the approach taken by authors of the AspectJ documentation and books by ignoring implementation issues. For the purpose of this paper, we are not interested in how code instrumentation is carried out, and in the practical constraints on which classes may or may not be instrumented. We similarly ignore the implementation of the matching between pointcuts and join points in AspectJ. Instead, we treat AspectJ as a black box, and examine its performance on carefully-chosen test cases.

2. CURRENT SEMANTICS OF ASPECTJ
The semantics of the wildcard operators (“*” and “..”) inside call and execution pointcuts are easily specified by considering them to be an abbreviation for the (infinite) union of all possible expansions. We will therefore ignore wildcards in the sequel. Also, in order to simplify the presentation, we will deal only with void functions of no arguments. This will entail no loss of generality. Since static methods are not inherited, we will also ignore those in the sequel.

2.1 Call Semantics
Consider the pointcut specified by \texttt{call(void A1.f())}. This should capture all calls to the method \texttt{f} defined in class \texttt{A1}. Indeed it does, but that is due to the careful wording of the previous sentence. What happens if \texttt{f} is inherited from another class? In order to answer this question, we will consider the following hierarchy of classes:

```java
public class A1 {
    public void f() {}
    public void g() {}
}

public class A2 extends A1 {
    public void h() {}
}

public class A3 extends A2 {
    public void f() {}
}
```

We then consider the following three variable definitions, in which the name of the variable indicates its static type and, if different, also its dynamic type:

\begin{itemize}
    \item \texttt{A1 s1 = new A1();}
    \item \texttt{A3 s3 = new A3();}
    \item \texttt{A1 s1d3 = new A1();}
\end{itemize}

It turns out that the pointcut \texttt{call(void A1.f())} captures the calls \texttt{s1.f()}, \texttt{s3.f()}, and \texttt{s1d3.f()}. Similarly, the pointcut \texttt{call(void A1.g())} captures the calls \texttt{s1.g()}, \texttt{s3.g()}, and \texttt{s1d3.g()}. It seems that even without the \texttt{+} modifier for, then?—but is consistent with the dynamic binding mechanism of Java. (We shall have more to say about the \texttt{+} modifier later.)

The pointcut \texttt{call(void A3.f())} captures the call \texttt{s3.f()} but not \texttt{s1d3.f()}. This implies that matching of call pointcuts is based on the static type of the variable, which is \textit{not} consistent with the dynamic binding principle, but may perhaps be justified based on the information available at the calling point. However, the real surprise is that the pointcut \texttt{call(void A3.g())} does not capture any join points in our example, not even \texttt{s3.g()}. The only difference between \texttt{f} and \texttt{g} in \texttt{A3} is that \texttt{f} is overridden whereas \texttt{g} is only inherited. Thus, it seems that for matching to succeed, it is necessary for the method to be lexically defined within the specified class—inheritance is not enough. We use the term “lexically defined” to indicate that a definition (first or overriding) of the method appears inside the definition of the class.

Thus we are led to the following model. The semantics of a pointcut will be given as a set of join points, formalized as a predicate specifying which join points are captured by the pointcut. Consider the following definitions:

- A pointcut \( pc_e = \text{call(void C.f())} \)
- A variable defined as \( S x = \text{new} D() \), and
- A join point \( jp = x.f() \).

That is, the pointcut specifies a class \( C \), and the target of the join point has the static type \( S \) and the dynamic type \( D \). (Obviously, \( D \) must be a descendant of \( S \) for this to compile correctly. We will denote this relationship by \( S \subseteq D \).) Then:

\[ jp \in pc_e \iff S \subseteq C \land f \text{ is lexically defined in } C. \]

### 2.2 Execution Semantics

Continuing with our example, we find that call and execution pointcuts capture exactly the same join points for \( s1 \) and \( s3 \) (we are ignoring other features of pointcuts, such as \texttt{this} and \texttt{target}). The only difference is in the treatment of \texttt{s1d3.f()}, which is captured by \texttt{execution(A1.f())} and \texttt{execution(A3.f())} but not by \texttt{call(A3.f())}. However, \texttt{execution(void A3.g())}, like the corresponding call pointcut, captures none of our join points. Thus, the rule for an execution pointcut

\[ pc_e = \text{execution(void C.f())} \]

seems to be:

\[ jp \in pc_e \iff D \subseteq C \land f \text{ is lexically defined in } C. \]

That is, the static type is replaced by the dynamic type. Again, this can be justified by the different type information available at execution join points, but is nevertheless an inconsistency in the semantics.

### 2.3 Subtype Pattern Semantics

The semantics of a subtype pattern such as \texttt{call(A1+.f())} should naturally be equivalent to the union of all possible expansions where \( A1 \) is replaced by any of its descendants. This is indeed the case in AspectJ. However, because of the surprising semantics described above, this has a subtle interpretation. If

\[ pc_e^+ = \text{call(void C+.f())} \]

is a call pointcut using subtypes, the matching rule is:

\[ jp \in pc_e^+ \iff S \subseteq C \land f \text{ is lexically defined in some } F \text{ s.t. } S \subseteq F \subseteq C. \]

In particular, the pointcut \texttt{call(A1+.h())} captures \texttt{s3.h()}, because \texttt{h} is defined in \( A2 \), but the same join point is not captured by \texttt{call(A3+.h())}, even though \( A3 \) has this method. This violates our expectation that \texttt{call(A3+.h())} should be a subset of \texttt{call(A1+.h())} that is identical for all join points in classes under \( A3 \).

Similarly, for

\[ pc_e^+ = \text{execution(void C+.f())}, \]

the matching rule is:

\[ jp \in pc_e^+ \iff D \subseteq C \land f \text{ is lexically defined in some } F \text{ s.t. } D \subseteq F \subseteq C. \]
2.4 Summary
The current semantics of AspectJ is summarized in Figure 1. It satisfies some of our intuitive expectations but violates others. The points on which AspectJ is consistent with the intuitive semantics are:

- Pointcuts with wildcards are equivalent to the union of all possible expansions.
- Pointcuts with subtype patterns are equivalent to the union of all pointcuts with subtypes substituted for the given type.
- The semantics of execution pointcuts is based on the dynamic type of the target.

On the following points the semantics of AspectJ deviates from our intuition:

- The semantics of call pointcuts is different from that of execution pointcuts, and depends on the static type of the target.
- Call and execution pointcuts only capture join points for classes where the given method is lexically defined.
- As a result of this, the difference between pointcuts with or without subtype patterns is subtle and unintuitive.

It is arguable whether pointcuts without subtype patterns should capture join points in subclasses at all. On the one hand, an instance of a class is ipso facto considered to belong to all its superclasses; this is reflected in the syntactic restrictions on assignment and parameter passing, and in the semantics of the instanceof operator. On the other hand, the existence of the subtype pattern modifier seems to imply the intention that a pointcut that does not use it refer only to instances of the specified class.

We believe that the lexical restrictions shown in these semantic definitions were unintended; their removal would greatly simplify the semantics. Some evidence that this is not the intended semantics comes from the following quote from one of the AspectJ gurus [4, p. 79]: “The [call(*) Account.* (...) pointcut] will pick up all the instance and static methods defined in the Account class and all the parent classes in the inheritance hierarchy” (emphasis added). This is not true in AspectJ, but is intuitively appealing.

Another interesting clue is the fact (pointed out to us by one of the anonymous reviewers) is that when the AspectJ compiler is invoked with the -1.4 switch, the set of joint-points defined by call pointcuts changes, and the restriction on the lexical definition of the method in the designated class is removed. Curiously, the behavior of execution pointcuts does not change even with this switch.

3. ALTERNATIVE SEMANTICS
If the current AspectJ semantics is inappropriate, we should propose one or more alternatives. As mentioned above, such alternatives should not restrict methods to be lexically defined in the designated class. Two questions remain:

1. should subclasses be included when the subtype pattern modifier does not appear in the pointcut; and
2. should call and execution pointcuts capture different join points.

These issues lead to four possible definitions of the semantics (see Figure 2). In these definitions we use the term “f exists in C” to denote the fact that the method f exists in class C, whether or not it is lexically defined (or overridden) in it. We use the term “broad” for those semantics that include subclasses even when subtypes are not indicated, and “narrow” for those that do not. The term “static” denotes semantics that use the static type for call pointcuts, and “dynamic” denotes those that use the dynamic type. It is important to note that although the join points captured by call and execution pointcuts are the same in the dynamic semantics, their properties (e.g., this and target) are different.

Each of the four semantics is consistent and reasonable. Perhaps the broad-dynamic semantics best reflects object-oriented principles, in that a reference to a class includes its subclasses, and the type that determines matching is the dynamic rather than static type of the variable. However,
The semantics of execution(void C+.f())

<table>
<thead>
<tr>
<th>Narrow</th>
<th>Broad</th>
</tr>
</thead>
<tbody>
<tr>
<td>$jp \in pc_c$ $\iff$ $S = C \land f \text{ exists in } C$</td>
<td>$jp \in pc_c$ $\iff$ $S \subseteq C \land f \text{ exists in } C$</td>
</tr>
<tr>
<td>$jp \in pc_s$ $\iff$ $D = C \land f \text{ exists in } C$</td>
<td>$jp \in pc_s$ $\iff$ $D \subseteq C \land f \text{ exists in } C$</td>
</tr>
<tr>
<td>$jp \in pc^+_c$ $\iff$ $S \subseteq C \land f \text{ exists in } S$</td>
<td>$jp \in pc^+_c$ $\iff$ $S \subseteq C \land f \text{ exists in } S$</td>
</tr>
<tr>
<td>$jp \in pc^+_s$ $\iff$ $D \subseteq C \land f \text{ exists in } D$</td>
<td>$jp \in pc^+_s$ $\iff$ $D \subseteq C \land f \text{ exists in } D$</td>
</tr>
</tbody>
</table>

(Static) (a) (b) (c) (d) (Dynamic)

Figure 1: Four possible semantics: (a) narrow–static; (b) broad–static; (c) narrow–dynamic; (d) broad–dynamic.

other semantics may be easier to use if they more closely reflect the intent of AspectJ programmers.

4. EXPRESSIVE POWER

The five semantic models presented above (current AspectJ semantics and four alternatives) are able to describe different sets of join points. However, AspectJ has additional pointcut designators, which may be used to modify the meaning of a pointcut. The question now is, what is the expressive power of each of the given semantics definitions? Are there meaningful sets of join points that can only be expressed by some of them?

The answer is, of course, positive. For example, a narrow semantics is easily expressed in the corresponding broad semantics. The pointcut call(void C.f()), whose meaning in the narrow–static semantics is “$S = C \land f \text{ exists in } C$” can be expressed in the broad–static semantics by the following pointcut:

call(void C.f()) && target(x) &&
if(x.getClass() == C.class)

However, the reverse is not true: in order to get a subset relation in the narrow semantics, we must use the subtype pattern modifier, but then there is no way to enforce the requirement that the method already exists in class C. So each broad semantics is strictly more expressive than the corresponding narrow semantics.

The static and dynamic semantics are incomparable. The dynamic semantics have no way of referring to the static type $(S)$, and the static semantics have no way of referring to the dynamic type $(D)$ in call pointcuts.

The semantics of execution(void C+.f()) in either of the dynamic semantics is easily expressed in the current AspectJ semantics by the expression

execution(void f()) && this(C).

The corresponding expression for call(void C+.f()) is
call(void f()) && target(C).

(Note that target in call pointcuts corresponds to this in execution pointcuts.) In order to understand the semantics of this expression under AspectJ, note that the call pointcut call(void f()) without a class designator is equivalent to call(Object+.f()), so when applying the semantics of Figure 1, the class inclusion condition is trivial, and we obtain simply that $f \text{ exists in } S$. Together with the additional requirement, target(C), we get that the semantics of the above expression in AspectJ is

$D \subseteq C \land f \text{ exists in } S$,

which is a little different from the dynamic semantics.

Under the current semantics, AspectJ has no way of requiring that $f \text{ exist in } C$ without being lexically defined in it. The alternative, going to the top of the inheritance hierarchy, then prevents the possibility of referring to the static type. On the other hand, the new proposed semantics have no way of requiring the lexical definition of a method in some class. (Note that the within and withincode constructs are too restrictive, because they do not capture overriding definitions. Also, these do not help with call pointcuts, because they refer to the caller code rather than the method implementation.)

Of course, the fact that one semantics is more expressive than another does not mean it is better. The question is what programmers (and automatic tools) really need to say. Furthermore, the cost of a complex semantics should be weighed against the convenience of the language. It might be better to adopt a simpler semantics for call and execution pointcuts, and add another construct to capture lexical definitions, if this is indeed necessary.

5. CONCLUSIONS

The current semantics of AspectJ has some unintuitive aspects. We have presented a number of alternative semantics, and compared their expressive power. The “right” semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in
practice. We hope that this paper will start a fruitful and constructive discussion on this question.

6. REFERENCES


Bibliography


תקציר

תוכנה ש搋 בנייה של השיטות

עוצב על פיו חוזה. ברר תכשיטי ייחודי לשתי ממגוון של תכשיטים Wolves ש errs את התוכן של התוכנה. בבר טענת, יישור תכשיטים באפקציה בין בשפה מתכתית. עובדה זו מפרידה את התוכן האפקטיבי החוזה שהしたこと באפקציה בין בשפה המתכתית לעיל פיו

הivable על יישור תכשיטי של האפקציה של התוכנה לאפקציה בין בשפה מתכתית. התוכנה מספקת lado לעיל פיו. בעיות אלו קיימות בקשר לתחום הדורשים, חוזה בו..., והן

הivable בתכשיטי שעון ב.pth את תבנית התחום של התוכנה בתחום התוכנות באפקציה בין בשפה מתכתית. עבודה בחלק

הivable בשפת הביצוען של שגרות בחרות של חתך נקודותשל הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של יחסי ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות зарегистри

תזכירים

הוהז והושקה בשתייה של התוכנה – הכותנה מנוגנת הביטים

הרשימה של ערכין קדומים וכרחים, הדורשים ממיקור בשתייה שגרותビュー, דוח

ה黪 התוכנה בתוכנה וביתריות שאופי של התוכנה של התוכנה. עבורה זו מפרידה את התוכן האפקטיבי בין בשפה מתכתית. בעיות אלו קיימות בקשר לתחום הדורשים, חוזה בו..., והן

הivable בתכשיטי שעון ב.pth את תבנית התחום של התוכנה בתחום מתכתית. עבודה בחלק

הivable בשפת הביצוען של שגרות בחרות של חתך נקודותשל הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות לבין חלופות ושגרות בחרות של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של ilişki ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חوزה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民间ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הסמנטיקה את חוקרים אנו AspectJ לירושה החתך נקודות של קישור ואת.

אנו חוזה, מחולקות מחולקות民間ן של חתך נקודות של הס먼...
תודה

בנרזני התודות ליצורותのごנותים של ל"ייר סמואל טיישברוביץ שילווה את גורתי במחקק

מראשות ותרם רבות עם הנותנים. ל"ייר ישים פלדמן על מקצתוות כללית והולך וול.

.baseUrl

תודה לappName, גישו חס וער稠וטו וחברת משמשים על מודלلهון חסרה וה плотה.

תודה לארץ ההברה, ייסו חס וער稠וטו וחברת משמשים על מודללהון חסרה וה плотה.

תודה לחברים לلزمונים: אודי בוקר, יאיר גורנשטיין, יזור אוסטרובר, גלעד ארנולד

וזר יחקיוו על התמידה בשעות קשות.

תודה להוריהם הקרים עם מלימהתombre בבריל.

תודה ליליה. אהובתי.
העבודה על יסודות העץ בוגר ההזזה

ע"וד היבט

(M.Sc.)

ה النوعוה חתול מתודרחשת לתואר מוסמך אוניברסיטת תל אביב, בהדרכת פרופ"ס ומצחוב

מאת

והד

העבודה הושגה בהדרחה על פרו"ס חורש

"יר יש פלדמן ו"יר שפורשל טישברוביצ"י

אב תשמ"ד