Universiteit Antwerpen

# Supporting Thread-local Aspects in a Virtual Machine for Modularized Crosscutting Concerns

**Tim Molderez**

Principal Advisor: Prof. Dr. Dirk Janssens

Assistant Advisor: Hans Schippers

Dissertation Submitted in May 2009 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

FoTS Formal Techniques in Software Engineering

# Acknowledgments

This thesis immediately starts off with the hardest section, the acknowledgments. A simple "thank you" somehow still manages to take a whole lot of effort from me compared to pondering on all these abstract concepts you're about to witness.

First of all, my thanks go out to Hans Schippers for providing continuous feedback and guiding this research in the right direction. I'd also like to thank Prof. Dr. Dirk Janssens for his reviews and giving me complete freedom on how to manage this thesis. Many thanks go out to my friends for answering the many curious questions I've had and for keeping me on my toes by frequently asking "And how is your thesis coming along?". One final thank you belongs to my family, for showing their support and letting me work quietly at the oddest of hours.

# Nederlandstalige samenvatting

Aspect-georiënteerd programmeren is een vrij recent paradigma dat de separation of concerns van programma's tracht te verbeteren door middel van een nieuwe constructie genaamd aspecten. Het aspect-georiënteerde paradigma is momenteel voornamelijk op het taalniveau gericht. Vele aspect-georiënteerde talen zijn gebouwd op een ander platform zoals een object-georiënteerde compiler, welk niet specifiek ontworpen is met dit nieuwe paradigma in het achterhoofd. Een meer elegante en flexibele oplossing bestaat uit het ondersteunen van het aspect-georiënteerde paradigma op een lager niveau. Deze thesis zal gebruik maken van delMDSoC, een virtuele machine die gericht is op paradigma's met ondersteuning voor "multi-dimensional separation of concerns" (MDSoC). Aspect-georiënteerd programmeren behoort tot deze groep van paradigma's. De minimale aspect-georiënteerde taal aj, afgeleid van de gekende AspectJ taal, is reeds geïmplementeerd in deze virtuele machine.

Een feature die niet beschikbaar is in zowel de aj taal en de delMDSoC virtuele machine zijn thread-local aspects, waarmee de programmeur het bereik van een aspect kan beperken tot één of meerdere threads. Deze thesis zal ondersteuning voor zowel multithreading en thread-local aspects introduceren in delMDSoC en aj.

De delMDSoC virtuele machine gebruikt een prototype-gebaseerd machine model als fundering. In dit model bestaat er enkel het concept van objecten die berichten naar elkaar kunnen sturen. Verschillende objecten kunnen ook aan elkaar gekoppeld worden via het delegation mechanisme, waarmee in essentie een ketting van objecten gevormd wordt. Als nu een boodschap wordt verstuurd naar een object en dit object begrijpt de boodschap niet, dan zal de boodschap worden doorgegeven aan het volgende object in de delegation ketting waar dit object toe behoort. Het proces herhaalt zich in dit volgende object en het blijft zich herhalen tot de ketting eindigt of tot een object gevonden is dat de boodschap kan afhandelen.

Het delegation mechanisme is tevens een zeer flexibel concept, doordat een object zelf eender welk object mag kiezen als opvolger in zijn delegation ketting. Hierdoor wordt het mogelijk om objecten dynamisch aan een ketting toe te voegen of te verwijderen. Het delegation mechanisme kan voor verschillende doeleinden gebruikt worden: Het kan gebruikt worden om het object-georiënteerde overervingsmechanisme mee te implementeren; er kunnen ook objecten in een delegation ketting tussengevoegd worden die een aspect voorstellen. Door de toevoeging van het delegation mechanisme wordt het dus mogelijk om verschillende paradigma's te emuleren boven op het prototype-gebaseerde machine model. Dit is reeds gebeurd voor het klasse-gebaseerde object-georiënteerde paradigma en het aspect-georiënteerde paradigma.

Deze ondersteuning voor andere paradigma's in combinatie met een implementatie van een zogenaamde "parsing expression grammar" parser generator maakt het mogelijk om op vrij eenvoudige wijze nieuwe talen te implementeren met delMDSoC. De taal waar deze thesis zich op zal concentreren is de aspect-georiënteerde aj taal.

Een eerste bijdrage van de thesis is het integreren van multithreading ondersteuning in de aj taal. Dit gebeurt door gebruik te maken van de Pthreads C library. Hiertoe worden prototype objecten toegevoegd die in essentie wrappers zijn rond de thread en mutex functionaliteit uit de Pthreads library, waarmee het mogelijk wordt om in delMDSoC threads aan te maken en te beheren. Het wrapper object dat een thread voorstelt kan ook rechtstreeks gebruikt worden in de aj taal alsof het een gewone klasse is. De multithreading ondersteuning in de parsing expression grammar van aj is tevens zodanig opgesteld dat multithreading voor de gebruiker op een gelijkaardige manier werkt als de Java taal, hoewel onderliggend Pthreads wor-

den gebruikt.

De tweede bijdrage van de thesis is het toevoegen van ondersteuning voor thread-local aspects aan de aj taal. Vier verschillende varianten op het thema van thread-local aspects worden bestudeerd:

- Het `cflow` pointcut: Dit heeft een inherent thread-local gedrag aangezien het de join points opvangt binnen een bepaalde control flow. Als een advice moet worden uitgevoerd, dan mag dit enkel gebeuren in de thread met de control flow die de opgevangen join points bevat.
- Thread-local aspects lokaal aan een thread klasse: In dit type thread-local aspects wordt het bereik van een pointcut beperkt tot thread instanties die een bepaalde thread klasse implementeren. Dit gebeurt door gebruik te maken van een nieuwe pointcut constructie genaamd `threadlocal`.
- Thread-local aspects lokaal aan een groep thread instanties: Dit zijn de thread-local aspects waar het bereik van een pointcut beperkt kan worden tot één of meerdere thread instanties; deze instanties worden aangeduid door middel van annotations.
- Per thread instantiatie: Dit is een aspect instantiatie strategie waarbij één aspect instantie wordt gemaakt voor elke thread. Elke aspect instantie mag dus enkel join points behandelen die zich bevinden binnen zijn overeenkomstige thread.

Deze thesis zal tonen dat het mogelijk is om deze verschillende types thread-local aspects te ondersteunen in de aj taal door de werking en implementatie ervan gedetailleerd te documenteren. De essentie van deze implementatie bestaat uit het aanpassen van de advice in een thread-local aspect zodanig dat een advice enkel uitgevoerd wordt wanneer het door de juiste thread wordt opgeroepen. Deze soort van meta-functionaliteit die aan een advice wordt toegevoegd noemt ook residual logic.

Het implementeren van thread-local aspects aan de hand van residual logic is een degelijke oplossing zolang het draait om één instantie van een aspect. Als er echter meerdere instanties met elk hun eigen residual logic achter elkaar worden geschakeld in een delegation ketting heeft dit een negatieve invloed op de performantie van het zoeken naar het object dat een bepaalde

boodschap implementeert. Dit is zeker het geval voor per thread instantiatie en kan ook een probleem vormen voor het `cflow` pointcut. Om soelaas te bieden op dit probleem wordt ook een alternatieve oplossing voorgesteld die het advice van aspecten ongemoeid laat en dus geen residual logic introduceert. In plaats van een lange ketting van op zichzelf staande objecten te maken waar elk object verantwoordelijk is voor één thread, is het beter om enkel één object tussen te voegen dat dynamisch kan kiezen welk thread-local object aan de beurt is. De residual logic die zich voordien in elk thread-local object bevond is in feite verhuisd naar dit ene extra object. Het performantieprobleem is nu ook opgelost doordat de lange ketting van objecten nu is vervangen door een korte ketting die zich dynamisch aan de huidige situatie aanpast.

Niet alleen is het nu mogelijk om gebruik te maken van thread-local aspects in de aspect-georiënteerde aj taal, maar ook is het mogelijk om de implementatie hiervan over te dragen naar andere MDSoC paradigma's. De implementatie van thread-local aspects speelt namelijk vooral in op het wijzigen van delegation kettingen en het implementeren van de juiste boodschappen om extra functionaliteit in te voegen. Deze handelingen hebben niets te maken met het aspect-georiënteerde paradigma, maar zijn specifiek aan het onderliggende prototype-gebaseerde model. Doordat delMDSoC ontworpen is voor een breed spectrum aan MDSoC paradigma's kan de implementatie van thread-local aspects vrij eenvoudig aangepast worden aan andere MDSoC paradigma's zoals context-georiënteerd programmeren.

# Abstract

Aspect-oriented programming is a recent programming paradigm that allows for an improved separation of concerns through a new construct called aspects. The aspect-oriented paradigm currently is mostly focused on the language level. Many aspect-oriented languages are built on top of another platform such as an object-oriented compiler, which wasn't designed with this new paradigm in mind. A more elegant and flexible solution would be to support the paradigm at a lower level. This thesis will make use of delMDSoC, a virtual machine with dedicated support for paradigms with "multi-dimensional separation of concerns". The aspect-oriented paradigm is the most commonly known example out of this group of paradigms. The minimal aspect-oriented language aj, which is a subset of the well-known AspectJ language, has already been implemented for the delMDSoC virtual machine.

One feature that isn't present in both the aj language nor the delMDSoC virtual machine are thread-local aspects, which allow the developer to limit the scope of an aspect to one or more threads. As there currently is no multithreading functionality present, this should be added first. This thesis will introduce and discuss the support for both multithreading and thread-local aspects in delMDSoC and aj.

# Contents

**1 Introduction** **1**

**2 Context** **3**

2.1 Aspect-oriented programming . . . . . 3

2.1.1 Separation of concerns . . . . . 3

2.1.2 The aspect-oriented paradigm . . . . . 4

2.1.3 Dynamic join points and join point shadows . . . . . 7

2.2 The delMDSoC virtual machine . . . . . 8

2.2.1 Machine model . . . . . 9

2.2.1.1 Message lookup . . . . . 9

2.2.1.2 Lieberman-style delegation . . . . . 11

2.2.1.3 Supporting the class-based object-oriented paradigm . . . . . 12

2.2.1.4 Supporting the aspect-oriented paradigm . . . . . 13

2.2.2 Implementation . . . . . 15

2.2.2.1 Combined Object-Lambda Architecture . . . . . 15

2.2.2.2 The delMDSoC kernel . . . . . 16

2.2.2.3 Parsing expression grammars . . . . . 16

**3 Multithreading support** **19**

3.1 Creating threads . . . . . 20

3.2 Synchronization and mutexes . . . . . 24

**4 Thread-local aspects** **26**

4.1 Applications . . . . . 27

4.2 Adapting the `cflow` pointcut construct . . . . . 28

4.3 Aspects local to a thread class . . . . . 32

4.4 Aspects local to thread instances . . . . . 36

4.5 Per thread instantiation . . . . . 39

4.5.1 Introductory solution . . . . . 39

4.5.2 Performance repercussions . . . . . 40

4.5.3 Modification of the message lookup procedure . . . . . 40

4.5.4 Making use of the flexibility of delegates . . . . . 42

**5 Related work** **45**

5.1 Concurrency models . . . . . 45

5.2 Thread-local aspects . . . . . 46

**6 Conclusion and future work** **49**

**A Modified aj parsing expression grammar** **51**

**B Glossary of acronyms** **62**

# List of Figures

2.1 Example of the message lookup procedure . . . . . 11
2.2 A class object and an instance . . . . . 13
2.3 An example of a class with an aspect . . . . . 14

4.1 Demonstration of the cflow pointcut . . . . . 31
4.2 Workings of a thread-local aspect that is scoped to a class . . . 35
4.3 Workings of a thread-local aspect that is scoped to instances . 38
4.4 Message lookup with the addition of the thread dimension . . 41
4.5 Using a switch object for thread-local aspects . . . . . 43

# List of Algorithms


2.1 A logging aspect in AspectJ . . . . . 6
2.2 Message lookup procedure in pseudocode . . . . . 10
2.3 An aj program demonstrating the observer pattern . . . . . 18
3.1 Creating and starting a thread in aj . . . . . 21
3.2 Partial implementation of the `Thread` class object . . . . . 23
3.3 Using the `synchronized` keyword . . . . . 24
4.1 Using the `cflow` construct . . . . . 29
4.2 Using the `threadlocal` construct . . . . . 33
4.3 Using annotations and the `threadlocal` construct . . . . . 37
5.1 CaesarJ's `deploy`-block construct . . . . . 47

# CHAPTER 1

# Introduction

Aspect-oriented programming (AOP) (16) is a fairly new programming paradigm that has gained popularity over the last couple of years. It attempts to improve the separation of concerns of programs by the introduction of a new construct called aspects, which allow the developer to separate so-called crosscutting concerns from other concerns. By separating these crosscutting concerns, it becomes easier to develop and maintain software. Most AOP languages are an extension to object-oriented programming (OOP) languages. The programs that are written in such AOP languages are typically executed by simply translating an aspect-oriented program back to the equivalent object-oriented program, which will then be compiled or interpreted on a platform that is designed for the object-oriented paradigm. Whilst such a solution takes less time in the short run, it's not the most elegant solution, as the generated object-oriented code tends to be quite verbose and it restricts the potential features supported by the aspect-oriented language. A more suitable solution to this problem would be to provide support for AOP at a lower level.

The delMDSoC virtual machine (VM), which will serve as the main working vehicle in this thesis, provides support for, among others, AOP at such a lower level, namely the machine level. Not only does delMDSoC offer dedicated AOP support, it supports the entire group of paradigms with "multi-dimensional separation of concerns" (MDSoC). This group of paradigms, of

which AOP is the most commonly known member, represents the paradigms that are specifically focused on improving applications' separation of concerns.

The aj language is one of the aspect-oriented languages that is implemented in the delMDSoC virtual machine. This minimal language actually is a subset of the well-known AspectJ (15) language, which is an aspect-oriented extension to the Java language.

The delMDSoC virtual machine currently has no support for a feature called thread-local aspects, which allow the programmer to confine an aspect's scope to certain threads. In other words, aspects can be set up such that they will only be active in a user-defined group of threads. Adding support for thread-local aspects to delMDSoC is the main goal of this thesis. Before this goal can be achieved however, the virtual machine should obviously be able to execute multiple threads concurrently. Unfortunately, no multithreading functionality is currently present in delMDSoC. Therefore, this thesis will also cover the implementation of multithreading in this virtual machine and the aj language.

Whilst there are other AOP languages that already offer thread-local aspects, such as CaesarJ , this thesis is considered a feasibility study. The reason for this choice is the fact that the prototype-based machine model that forms the foundation of the delMDSoC virtual machine is quite different from alternative solutions that already support thread-local aspects. Therefore, providing thread-local aspects is a problem that has not been solved yet within the realm of this machine model.

The outline of this thesis is structured as follows: Chapter 2 will introduce the various concepts that are used within this thesis in more detail. Chapter 3 will discuss how support for multithreading was added to the delMDSoC virtual machine. Chapter 4 represents the core of the thesis, as it documents the details of adding support for thread-local aspects. Chapter 5 will then compare the implementation of thread-local aspects in the aj language with other approaches in other languages and virtual machines. Finally, chapter 6 concludes this thesis and also presents potential paths that might be taken in the future.

# CHAPTER 2

# Context

Before diving into the details of thread-local aspects, an introduction is given to the various concepts and tools that are used within this thesis.

## 2.1 Aspect-oriented programming

### 2.1.1 Separation of concerns

Whilst there is no clear-cut definition for what a *concern* exactly is or isn't, it usually is considered synonymous to a feature or behaviour. The term separation of concerns (SoC) (21) denotes the process of dividing a program into several different modules such that their functionality overlaps as little as possible. In other words, separation of concerns tries to split up a program into modules such that each module ideally only implements one single concern.

By introducing the notion of classes, the object-oriented paradigm has improved on the separation of concerns of programs when compared to the

older procedural/imperative paradigm. However, object-oriented programming isn't capable of separating so-called *crosscutting concerns* from other concerns. A crosscutting concern is a concern that is usually spread throughout several different places of a program's modules and is intertwined with other concerns. Typical examples of crosscutting concerns are logging, caching, authentication and authorization.

### 2.1.2 The aspect-oriented paradigm

AOP (16) is a fairly recent programming paradigm that allows the developer to separate crosscutting concerns from other concerns by means of a new construct called *aspects*. AOP is designed to sit on top of another paradigm, much like the object-oriented paradigm is an extension of the procedural paradigm. In most cases, aspect-oriented languages extend from an object-oriented language. [1] The most well-known AOP language is AspectJ (15), which extends the object-oriented Java language.

A good way to explain the concept of AOP is by means of an example. Suppose there is a banking application that keeps track of its customers' actions by maintaining a log. In a traditional object-oriented implementation, the code that makes a call to write a new entry into the log is usually located within the code that implements a banking operation. There are two reasons to suspect this logging functionality is a crosscutting concern:

- Because the business logic of a banking operation is completely irrelevant to logging functionality, two different concerns are now intertwined/tangled together by putting the call that accesses the log within the code of a banking operation.
- The call to the log is present within every single bank operation and presumably in other locations of the banking application as well. In other words, logging functionality is also scattered throughout several different locations in the application.

These two symptoms lead to the conclusion that logging is indeed a crosscutting concern within the banking application. In its current object-oriented form, the logging crosscutting concern impacts the application in two ways:

[1] There also are AOP languages that, for example, extend functional languages (3) and even query languages (4).

- Because the logging concern cross-cuts other concerns, the source code becomes more difficult to understand.
- Because logging is scattered over several places within the program, the code is more prone to errors whenever the interface to the log should change.

In order to resolve these problems, the aspect-oriented paradigm introduces a new construction called aspects, which can be used to encapsulate crosscutting concerns. An aspect mainly consists of two components: *pointcuts* and *advice*[2], whereas each advice is associated with a pointcut.

- A pointcut is an expression that determines *when* an advice must be executed. More specifically, a pointcut is a set of so-called *join points*, which represent locations in a program's execution graph.
- In essence, an advice is a block of code that encapsulates (part of[3]) the behaviour of a crosscutting concern. This code will be executed whenever the corresponding pointcut is triggered. A pointcut is triggered/activated whenever a program's execution reaches a join point that belongs to the set of join points specified by the pointcut. If a pointcut is triggered, there is a choice between three kinds of advice:
  - Use a *before* advice: Whenever a pointcut is triggered, the corresponding advice will be executed *before* program execution continues at the join point that was reached.
  - Use an *after* advice: Whenever a pointcut is triggered, the corresponding advice will be executed *after* the functionality at the current join point has been executed.
  - Use an *around* advice: Whenever a pointcut is triggered, the corresponding advice is executed. The advice itself can then decide when the functionality at current join point should be executed. This is usually done by inserting a "proceed" statement in the advice. If desired, the advice can even decide to skip the functionality at the current join point.
    The before and after advice actually are special cases of the around advice:

[2]Keep in mind that the plural form of the word "advice" is "advice" as well.

[3]Because an aspect can contain multiple advice, each advice represents a part of the behaviour of the crosscutting concern that the aspect is implementing.

- A before advice is an around advice that ends with a proceed statement.
- An after advice is an around advice that starts with a proceed statement.

In the banking example, the crosscutting concern that represents logging functionality can now be separated from the other concerns by making use of aspects. First, simply remove all logging functionality from the application. This functionality can then be reinserted into the application by introducing one new aspect. The single advice that is present in this aspect would be a block of code that writes a new entry into the log. If, for example, the completion of bank operations should be entered into the log, using an after advice makes a suitable choice. The aspect's pointcut would be an expression that describes all calls to the bank operations that must be logged.

**Algorithm 2.1** A logging aspect in AspectJ

```
1 public aspect Logging {
2     // Advice type and pointcut
3     after(TransactionData tD): execution(* do*(TransactionData) && args(tD))
4     // Advice
5     {
6         log.write("Operation completed: " + tD.toString());
7     }
8 }
```

Algorithm 2.1 shows how the logging aspect could be implemented in the AspectJ language. The pointcut that is shown can be interpreted as follows: "Capture all join points where a method is executed of which the name starts with 'do' and which has one parameter of type `TransactionData`. Also bind this parameter to the local variable `tD`." In other words, this pointcut assumes all banking operations start with "do" and have one parameter. Whilst these assumptions may become invalid because of a change that was made during maintenance, it now is clear that all logging functionality is located at one single place in the source code, which makes the code easier to understand and maintain than its object-oriented equivalent.

When the application is executed, at some point in time, the code of an aspect's advice must be inserted at the join points where it will be applied, which could happen at compile-time, at runtime, or even somewhere in between. This process is called *weaving*; it reintroduces the scattering and

tangling behaviour of crosscutting concerns into the application in order to make them executable[4].

In the end, the essence of AOP is to turn the explicitness of crosscutting concerns in OOP into implicit behaviour by making use of aspects. Whereas functions and methods should be called explicitly by whoever wants to make use of their functionality, aspects decide for themselves where and when they should offer their functionality. Therefore, the application of an advice can be seen as an implicit function call. Making crosscutting concerns implicit has both advantages and disadvantages. The application becomes easier to maintain because there now is a better separation of concerns. On the other hand, whilst the crosscutting concerns seem to be completely separated from other concerns, they will still crosscut other concerns in the end, which becomes less obvious for the developer making use of AOP.

### 2.1.3 Dynamic join points and join point shadows

Section 2.1.2 has briefly touched the notion of join points. This section will provide a more in-depth notion of what a join point is. The definition of a join point is as follows: A join point is a location in the execution graph of a running application. In other words, it is a point in an application's control flow. This definition implies that join points only become visible at runtime. Therefore, a join point is an inherently dynamic entity.

To stress the dynamic nature of a join point, they are also called dynamic join points (31) or virtual join points (12). When the prefix "virtual" is added, the name also hints at the fact that join points are related to virtual method dispatch. When a virtual method call is made, it can only be decided at runtime which method should actually be called, depending on the dynamic type of the instance that implements the method. Likewise, when a virtual join point is reached, it can only be decided at runtime which advice, if any, should be executed.

When compiling aspect-oriented applications, the notion of *join point shadows* comes into play. Every dynamic join point of an application can be mapped onto a static location in the application's source code. During compilation, additional code will be inserted at these join point shadows in order to make the application's aspects functional.

[4]Because the readability/maintainability/... of the resulting machine code is of little concern to a human being, it does not matter that the crosscutting concerns are scattered all over the application once again.

This is analogous with the notion of a breakpoint used in debuggers: A join point shadow corresponds with the locations in the source code that can be marked with a breakpoint. During compilation, extra code will be inserted at these breakpoints, such that runtime information can be fed back to the debugger. At runtime, a single breakpoint can be triggered at multiple locations in the application's execution graph; these locations then correspond to join points. [5]

## 2.2 The delMDSoC virtual machine

Most research related to the aspect-oriented paradigm is currently situated at the language level. Many implementations of aspect-oriented languages simply convert all aspect-oriented features back to object-oriented ones, such that an already existing object-oriented compiler or interpreter can be reused to execute aspect-oriented applications. While this approach works perfectly and is a quick way to implement an aspect-oriented language, this existing object-oriented technology was never designed with AOP in mind, resulting in a verbose object-oriented representation. A more elegant solution would be to provide support for the aspect-oriented paradigm at a lower level. The delMDSoC virtual machine [6] (25) provides such support at the machine level, as it implements a machine model that is suited for AOP. Initially, delMDSoC was called delAOP because it was solely focused at the AOP paradigm. However, it was later renamed to delMDSoC to stress the fact that it can also handle a group of paradigms with support for "multi-dimensional separation of concerns" (MDSoC) (29). The group of MDSoC paradigms are all aimed specifically at improving the separation of concerns of applications; AOP currently is the most well-known member of MDSoC paradigms.

[5] Because of the similarities between join points and breakpoints, it shouldn't come as a surprise that aspects can be used for debugging (30) and profiling purposes as well.

[6] At the time of writing, the delMDSoC virtual machine is available at: http://www.hpi.uni-potsdam.de/hirschfeld/projects/delmdsoc/index.html

### 2.2.1 Machine model

The machine model (12) that is implemented in delMDSoC is *prototype-based*, which is a style of object-oriented programming. Unlike the typical class-based object-oriented approach, prototype-based programming does not know a concept of classes. In this paradigm, there only exist objects. Similar to the instancing mechanism in class-based languages, new "instances" of an object in a prototype-based language are created by cloning the object. Objects can communicate by sending messages to one another, which is quite similar to calling methods. The only entities inside an object are message implementations, the prototype-based equivalent of method implementations. If a particular message is sent to an object that has a matching message implementation, it will reply to this message with another object, which then is the equivalent of a method that returns an object.

The main advantage of prototype-based programming is its simplicity. However, this does not make the paradigm any less powerful than class-based programming, as it will be shown in section 2.2.1.3 that the class-based paradigm can be emulated using prototypes.

#### 2.2.1.1 Message lookup

Message lookup in delMDSoC's machine model is partially similar to virtual method lookup in a class-based model. When calling a virtual method on an object, virtual method lookup first checks whether this method is present in the class representing the dynamic type of this object. If not, the superclass is checked. If the method implementation is not found there, check the superclass of this superclass. Virtual method lookup simply continues to follow the object's inheritance chain until an implementation of the requested method is found.

The *message lookup procedure* for delMDSoC's prototype-based machine model is shown as pseudocode in algorithm 2.2. The procedure first starts with sending a message to an object; this object is called the receiver. If the receiver does not contain a message implementation for the message that it received, the receiver will send the "parent" message to itself. Every object must provide an implementation for this special parent message; it will return the "parent object" of an object. Just like virtual method lookup, message lookup will follow an object's inheritance chain by sending the parent message to the object, then its parent object, then the parent object of this

**Algorithm 2.2** Message lookup procedure in pseudocode

```
1  currentParentObject = receiver;
2  currentDelegateObject = receiver;
3
4  while (currentDelegateObject != null) {
5      if (currentParentObject has implementation for received message) {
6          let currentParentObject handle the message
7          and return the resulting object to the sender;
8      } else {
9          // Move to the next object in the inheritance chain
10         currentParentObject = send "parent" to currentParentObject;
11         if (currentParentObject == null) {
12             // Move to the next object in the delegation chain
13             currentDelegateObject = send "delegate" to currentDelegateObject;
14             currentParentObject = currentDelegateObject;
15         }
16     }
17 }
18
19 return error "the receiver does not understand this message";
```

parent object and so on... until an implementation is found for the message that was sent to the receiver.

However, it is possible that the inheritance chain stops at some point and that the appropriate message implementation still hasn't been found. Unlike virtual method lookup, the story does not end here by displaying some error message. Message lookup will still continue its search by sending the special "delegate" message to the starting object, which is the receiver. This "delegate" message must also be implemented by all objects; the "delegate" message must return a *delegate object*, which can be any object where messages are delegated to in case the inheritance chain fails. Just like the inheritance chain, it also is possible to create a chain of delegate objects by using the "delegate" message. Message lookup can now proceed its search at the delegate object. It will first follow the inheritance chain of this delegate object and then send the "delegate" message to the delegate object, where the whole process repeats itself until the appropriate implementation is found. Only if the inheritance chain of the last object in the delegate chain fails, no one has implemented the requested message and an error message should be shown.

Figure 2.1 shows an example of the message lookup procedure, which is started by sending the message "foo" to object a. Message lookup will search objects a, b, c, ... until object h is found, where the message implementation of "foo" is located.

By adding this delegation chain, another dimension is added to the message

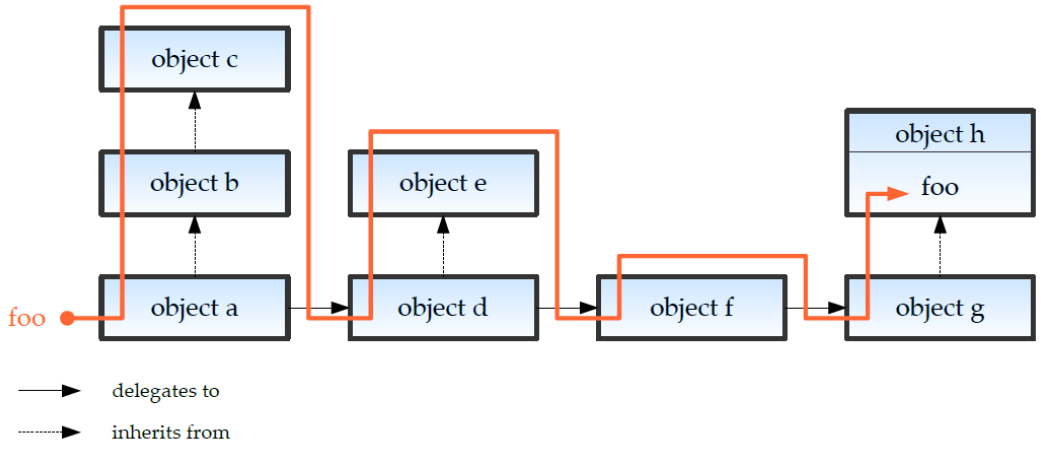


Figure 2.1: Example of the message lookup procedure

lookup procedure. What hasn't been stressed yet is that an object's message implementation for the "delegate" message is a function. This implies that the link between an object and its delegate object is not fixed. An object may choose whoever its delegate will be depending on, for example, the object's current state. The flexibility of the delegation chain can be put to good use in several different manners: In section 2.2.1.4 it will be used to implement the aspect-oriented paradigm; in chapter 4 it is used to implement thread-local aspects.

#### 2.2.1.2 Lieberman-style delegation

The message lookup procedure that was explained in the previous section was specific to delMDSoC's machine model; other prototype-based models may or may not have support for delegation or have another style of delegation. The style of delegation that is used in delMDSoC is called "Lieberman-style delegation" (23). It is characterized by the following two properties:

**multi-dimensional dynamic bind** This term refers to the fact that the "delegate" message is dynamic and can return any arbitrary object. Therefore, several different objects can be an object's delegate object, resulting in multiple delegation chains.

**receiver splitting** In the class-based object-oriented paradigm, instances can refer to themselves by making use of the `this` keyword. In the prototype-based paradigm, the keyword `self` is used instead. What is special

about Lieberman-style delegation is that there are two kinds of `self` that must be taken into account: The first `self` is used to send messages to yourself; the second `self`, also called `stateful_self`, is used to access your own state. Because of the addition of the delegation mechanism, the two selves can diverge. `self` is a fixed object and is always set to the first object in the delegation chain, which is the receiver. `stateful_self` is set to the current object of the delegation chain; the object where the message implementation was found is thus located in `stateful_self`'s inheritance chain. In the message lookup example that was shown in figure 2.1, object a is `self` and object g is `stateful_self`.

#### 2.2.1.3 Supporting the class-based object-oriented paradigm

In order to implement a class-based object-oriented language using the delMD-SoC VM, a class-based model must be present. Such a model can be emulated with relative ease by using the prototype-based machine model as the foundation.

The concept of a class is introduced by using "class objects", which are nothing more than plain objects. A class's methods are implemented by adding message implementations to the class object.

Class instances also are plain objects; these instances only contain state in the form of a series of getter and setter message implementations that can access and modify the instance's fields. However, it also is perfectly possible to call methods from the instance object because the class object is set up as the instance's delegate object. Therefore, when methods are called on instances, the message lookup procedure makes sure the message ends up being delegated to the class object. An example of a class and its instance is shown in figure 2.2, where the instance of class A links to class A itself with the delegate message.

Support for inheritance must be added as well. There are a couple of ways to add this support: One could make use of parent objects, as these form the inheritance chain between objects, which was shown in section 2.2. Another option is to copy everything from a superclass object into the subclass object. In this sense, subclasses inherit everything from their superclass as soon as they are constructed. The latter option was implemented in delMD-SoC, which implies the inheritance chain simply is unused! Only the delegation chain remains, thereby significantly simplifying the message lookup

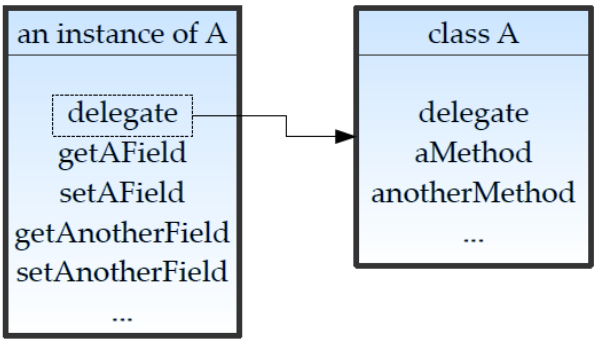


Figure 2.2: A class object and an instance

procedure.

All of the features of a class-based object-oriented model are now present; the following section will continue from this point and add extra features on top of the current model in order to provide support the aspect-oriented paradigm.

#### 2.2.1.4 Supporting the aspect-oriented paradigm

As was mentioned in section 2.2.1.1, the delegation chain will play an important role in adding support for the aspect-oriented paradigm. Using delMDSoC's machine model, it becomes possible to dynamically insert and remove aspects from an application. This is done by altering an object's delegation chain at runtime. If an aspect's pointcut is trying to match all calls to a certain method from a certain class, the aspect can be implemented by inserting a new object into the delegation chain of the class object, such that the class object becomes the delegate of the new aspect object.

The problem that now occurs is that all class instances still have the class object as their delegate. In order for the newly added aspect object to work properly, all instances should have this aspect as their delegate. Otherwise message lookup will never visit the aspect when messages are sent to the class object. Readjusting all of these instances can easily turn out to be quite a large chore. A better solution would be to have a fixed access point to a class object, which will be used as the delegate for all instances of this class. If the class object's delegation chain should change, this fixed access point should still function properly. The solution implemented in delMDSoC indeed provides such fixed access points for each class object. These access

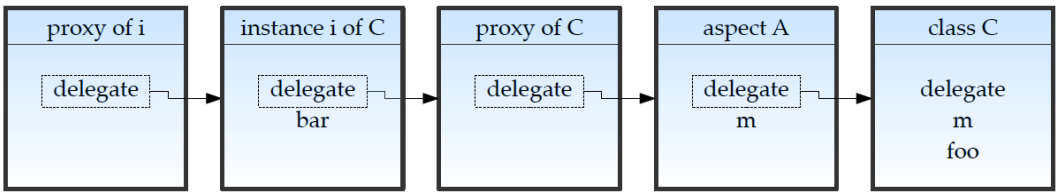


Figure 2.3: An example of a class with an aspect

points are called *proxy objects*; these are objects that only contain a message implementation for the "delegate" message. This means a proxy will delegate all the messages that it receives. Initially a proxy's delegate object is its corresponding class object. However, when an aspect is inserted, it will be inserted in between the proxy and the class object. The benefit of using these proxy object is that instances can always reliably set their delegate to the proxy object without ever having to readjust it, even if the class object's delegation chain is altered.

Whereas aspects are typically applied to all instances of a class, this aspect-oriented model also allows aspects to be applied on specific instances only. This is possible because not only do class objects have a proxy, all instance objects as well all receive their own proxy object. Therefore, aspects can be inserted in between an instance and its proxy, such that the aspect only affects this specific instance.

Figure 2.3 shows an example of a delegation chain where an aspect is inserted between class C and its proxy. The fact that aspect A has a message implementation for the method m implies that the aspect's pointcut is matching all calls to method m in class C. Whenever someone calls this method m on instance i, message lookup will start at the instance's proxy, since each object that wants to access i must go through its proxy first. Before message lookup can reach the method's implementation in class C, aspect A will intercept because of its own implementation of m. The aspect's implementation of m will execute the advice and also make a call to the original implementation of method m in class C. Whether this call is made before, after or within the execution of the advice depends on the advice type (before, after or around advice).

### 2.2.2 Implementation

#### 2.2.2.1 Combined Object-Lambda Architecture

The delMDSoC virtual machine was implemented on a "Combined Object-Lambda Architecture" (COLA), which is "a new way to construct programming languages, systems, environments and applications" (22). This project aims to create a "programming language substrate", which means it is a platform that aims to simplify implementing programming languages in a wide range of paradigms. Many, if not all, paradigms can be implemented on the platform because everything in COLA is late-bound [7].

COLA consists of two components: The first is the prototype-based model that was discussed in 2.2.1, also called the *representation layer*. This representation layer also provides a Smalltalk-like (9) language that makes use of the prototype-based model. The second component is a functional Scheme-like (10) language using S-expressions [8] (17), which can be transformed into an executable form. This component also is called the *meaning layer*. This Scheme-like language provides the desired end-user platform, as everything in it is late-bound. The JIT compiler[9] that compiles the language was first implemented using the Smalltalk-like language from the representation layer. Afterwards the compiler was reimplemented in its own language, such that the system became self-describing.

The main selling points of a COLA are the following:

**Simplicity** As mentioned before, the key of a prototype-based model is its simplicity. As a result, the model is very general-purpose and is easily extensible.

[7] If something is late-bound, it means it is decided/bound dynamically, at runtime. For example, C++'s virtual methods are late-bound. The main advantage of late binding is flexibility, whilst the main disadvantage is that it becomes more difficult to achieve good performance, when compared to early binding.

[8] Symbolic expressions or S-expressions are a representation of data in human-readable text form, mostly composed of symbols and lists. They are typically known from functional languages such as Lisp or Scheme, where S-expressions form the language's building blocks.

[9] A just-in-time (JIT) compiler is a program that sits somewhere in between a compiler and an interpreter. It compiles chunks of source code as soon as they need to be executed, which all happens at runtime. Compared to a regular interpreter, a JIT compiler has a performance benefit because the compiled chunks of code can be cached. Compared to a regular compiler, a JIT compiler has the advantage that it can recompile chunks of code if needed. For example, if a particular chunk of code is referred to a large number of times, it could be recompiled with more optimization options enabled.

**Openness** All parts of the whole COLA system are visible and open for modification and extension.

**Evolutionary programming** COLA is a meta-circular system, which means the system is described in terms of itself. The system has bootstrapped itself from a C library until the compiler for the functional Scheme-like language from the meaning layer is written in the same language. This allows the system to evolve from within to create the final end-user system.

**User-centered construction** The user isn't bound to any restrictions. Programming languages, models, message sending mechanisms, ... Everything can be suited to the user's needs.

#### 2.2.2.2 The delMDSoC kernel

The kernel of the delMDSoC VM is implemented as a set of macros written in COLA's meaning layer, which aid in defining classes, aspects, methods, ... These macros essentially implement the class-based and aspect-oriented programming paradigms, as described in sections 2.2.1.3 and 2.1.2.

Before these macros could be implemented however, the representation layer also had to be modified in order to provide support for proxified objects [10] and Lieberman-style delegation. Implementing Lieberman-style delegation is an example that demonstrates the openness and flexibility of a COLA, as both the message sending and lookup mechanisms had to be overwritten in order to get this new style of delegation to work.

#### 2.2.2.3 Parsing expression grammars

A *parsing expression grammar* (PEG) (6; 7) is a fairly recent type of formal grammar that can be used to describe the of syntax programming languages as a set of grammar rules. Current parser generators such as yacc or ANTLR typically make use of context-free grammars. The main disadvantage of such context-free grammars is that these are prone to ambiguity, which is to be avoided at all costs, especially in a programming language. This implies that such grammars must be carefully tweaked in order to be able to prevent or resolve ambiguities. A PEG does not have this disadvantage, as it cannot introduce any ambiguities. Another advantage of a PEG is that its parser does not need to break up its input into a set of tokens as a separate step.

[10] Proxified objects simply are the objects that have a proxy object attached to them.

A parser generator that makes use of parsing expression grammars was implemented on top of COLA. In this implementation, a PEG is defined in the form of a .peg file, which is used to define both the syntax and the semantics of a programming language.

The macros that are provided by the delMDSoC kernel can be used within a .peg file, enabling the user to easily define his own class-based object-oriented and aspect-oriented languages. Several languages have already been implemented by using a PEG in combination with the delMDSoC kernel:

**j** A minimal subset of the object-oriented Java language

**ij** An extension of the j language that includes intertype declarations[11]

**aj** A minimal subset of the aspect-oriented AspectJ language

**cj** A minimal context-oriented (14) language

These languages are all minimal for two reasons: Firstly, they are proofs-of-concept that several different languages of various programming paradigms can be implemented on top of the prototype-based object model. Secondly, the languages are also accompanied by descriptions that formally describe the languages' operational semantics (27; 26). To keep these formal descriptions free from redundancy and as simple as possible, the languages typically are set up to be as minimal as possible.

The aspect-oriented aj language will be used in the following chapters to add support for multithreading and thread-local aspects. Its .peg file is shown in appendix A. (All syntax rules are shown in this appendix, but the semantics of several grammar rules that aren't relevant to multithreading or thread-local aspects are hidden.) It may be interesting to have a quick look at this appendix before continuing, as it shows how PEGs work in practice; it also contains some guidelines on how to read a PEG.

[11] *Intertype declarations* or *introductions* are field or method declarations that aren't located in the class that is the owner of these fields or methods. This feature also is present in AOP and can be used to introduce new fields or methods into a class, where the declaration is located in an aspect (or another class). An introduction can be accessed by the aspect or class that declares it, but usually not by the class that is the owner of the introduction.

**Algorithm 2.3** An aj program demonstrating the observer pattern

```
1  class Subject ext ProxifiedObject {
2      Integer attr
3  }
4
5  class X ext ProxifiedObject {
6      Subject s
7
8      void main (X x) {
9          x.s := new Subject;
10         x.s.attr := 50;
11         x.s.attr := 25;
12     }
13 }
14
15 aspect Observer {
16     before set(Integer Subject.attr){
17         if (v.neq(r.attr)){
18             out.println("attr changed")
19         } else {0}
20     }
21 }
```

Algorithm 2.3 shows an example program written in the aj language, which can be fed as an input file to the aj PEG, who will then parse and execute the file. The example program shows how AOP can be used to implement the observer design pattern (8). The `Observer` aspect will print a message just before `Subject.attr` changes. In this case, `Subject.attr` changes twice , which means the message is printed twice as well.

Finally, when reading the aj examples in this thesis, a couple of oddities (that are mainly due to the language's simplicity) should be kept in mind:

- All methods must have exactly one parameter called `x`, whether it is used or not.
- There are no local variables; the only available variables are a class's or aspect's fields and the method's parameter `x`.
- The main class is called `X`.
- There is no type checking; if a method's parameter is unused, the nonexisting type `Dummy` is used.
- The syntax of aj is similar to AspectJ's, but is not an exact subset.

# CHAPTER 3

# Multithreading support

The main goal of this thesis is to add support for thread-local aspects to the delMDSoC virtual machine. However, as neither COLA nor the delMDSoC VM contain any multithreading functionality, this functionality should be added first.

Essentially there are two choices when trying to add multithreading support to delMDSoC: Either make use of an already existing threading library such as POSIX threads (19) or extend delMDSoC's machine model by implementing a new concurrency model that is perfectly suited to delMDSoC. Both approaches have their own set of advantages and disadvantages:

**Advantages and disadvantages of using an existing library:**

+ Implementing this solution will presumably take less time than implementing something from scratch.

+ An existing multithreading library such as POSIX threads is very stable as it has already been tested thoroughly, both by the library's developers and on the field.

+ An existing library will also be more complete in terms of features compared to a completely new library.

- Using an existing library will not provide a solution that is as elegant as writing a new library. In case of POSIX threads, its procedural implementation (because of the C language) would be forced to work with the prototype-based machine model.

- If the existing library is more platform-specific than COLA, the whole system will become less portable. Currently COLA compiles the code from the representational and meaning layer to the C language, which the developers of COLA consider a "portable high-level assembly language" (22). COLA has already been tested on Windows and several Linux flavours.

The advantages and disadvantages of extending the machine model are the exact opposite of using an existing library: It will take more time to implement; it will probably will not be as stable as a thoroughly tested library, but it will be better suited to the prototype-based machine model.

Weighing the advantages against the disadvantages resulted in a preference for using the POSIX threads library, also called the Pthreads library. The deciding factor simply was the smaller amount of time that is needed to integrate this library into delMDSoC. As the C language forms the underlying "assembly language" of COLA and because COLA's openness allows access to every part of the platform, it should be feasible to integrate the Pthreads library. The Pthreads library was chosen in particular because it is the de facto standard for adding multithreading support on Unix-like platforms such as Linux, Mac OS X and Solaris. A Windows port also is available, which implies the level of portability of COLA remains untouched. The only remaining disadvantage is the fact that this library should force its way into the prototype-based paradigm that is the foundation of COLA. In concrete terms, this implies a lot of C type casting, which is not considered very good practice.

## 3.1 Creating threads

The ability to create and start a new thread is first added to the aj language. Because aj is a subset of AspectJ, which in turn is a Java extension, creating

threads in aj is done in the exact same fashion as the Java language: Subclass the `Thread` class and override the `run` method to implement the thread's behaviour. The thread can then be started by creating an instance of the new subclass and calling its `start` method. An example aj application is shown in algorithm 3.1 where a thread is created and started. The main thread will also wait for the new thread to end by making use of the thread's `join` method.

**Algorithm 3.1** Creating and starting a thread in aj

```
1  class PrintHelloThread ext Thread {
2      void run(Dummy x) {
3          out.println("Hello world")
4      }
5  }
6
7  class X ext ProxifiedObject {
8      PrintHelloThread thread
9
10     void main(X x) {
11         x.thread := new PrintHelloThread;
12         x.thread.start(0);
13         x.thread.join(0);
14     }
15 }
```

All of this functionality can be implemented without even touching aj's parsing expression grammar. Whilst it was first implemented here as a first attempt, it would make more sense to implement this functionality within the COLA representation layer, using its Smalltalk-like language. It doesn't quite belong in aj's .peg file because this functionality does not modify the language's syntax; the `Thread` class is only added. This class was added to the representation layer by defining a new object that has `ProxifiedObject` as its parent object. By doing so, the object automatically becomes as a class object, which can be used in aj applications as if it were a regular class.

The `Thread` object essentially is a wrapper around a Pthread handle, which can be used to identify and control a particular thread. Suppose an instance is created of class `MyThread`, which is a subclass of `Thread` defined by the user, the following steps will be taken when calling its start method:

1. The message "start" is sent to the instance object representing the `MyThread` instance; message lookup will eventually find the implementation of this message within the `Thread` class object.

2. Inside the "start" message implementation, the message implementation "startHelper" will be started in a new Pthread. The only responsibility of "startHelper" is to send the "run" message to the `MyThread` instance, which will execute the new thread's behaviour. One may wonder why "startHelper" is needed and the "run" message isn't started directly. The reason for this is that message implementations end up as C functions that require several parameters, such as `self` and `stateful_self`. Unfortunately, the functions that can be started in a new thread using Pthreads' `pthread_create()` must comply to a specific interface that only has one parameter. Therefore, the message implementation of "startHelper" will be cast to a function pointer that has this correct interface. This cannot be done for "run" since this message implementation is defined by the user in `MyThread` and he does not know that his implementation is not called in the usual way. This is why "startHelper" must make sure that the "run" message is sent with the regular message sending mechanism.

3. The C function `pthread_create()` is called such that the "startHelper" message implementation is called in a new thread and receives one parameter, being the `self` pointer of the `MyThread` instance object. This parameter is crucial to enable "startHelper" to send messages to the `MyThread` instance object.

4. Within "startHelper", which is now running inside a new thread, the `self` pointer is used to send the message "run" to the `MyThread` instance object. Message lookup will end up finding the run method of the `MyThread` class and will execute it. In other words, the run method of the `MyThread` instance is now running in its own thread.

The steps described above can also be seen in algorithm 3.2, which shows a part of the implementation of the `Thread` class object. Whereas the language of COLA's representation layer is a variant of Smalltalk, it also is possible to insert plain C code into message implementations by writing it within curly brackets. When the Smalltalk code is compiled to C, these blocks of inserted C code simply remain unaltered. Therefore, the inserted blocks of C code depend on the generated C code that is produced by the compiler. This means that the inserted blocks of C code are fragile and could break if the compiler is modified. It can be seen in algorithm 3.2 that some assumptions were made about the generated C code. For example, each

Smalltalk variable is prefixed with "v_" in the C code. It can also be noticed that the Smalltalk code is dynamically typed, which results in C code that always uses the type `oop`, which represents a prototype object.

**Algorithm 3.2** Partial implementation of the `Thread` class object

```
1  { import: ProxifiedObject }
2  { include "pthread.h" }
3
4  Thread : ProxifiedObject (handle)
5
6  " Constructor "
7  Thread new [
8      self := super new.
9      { self->v_handle = 0; }
10 ]
11
12 " Starts 'run' in a new thread "
13 Thread start {
14     void* (*funcPtr)(void*) = (void* (*)(void*)) Thread__startHelper;
15     self->v_handle = malloc(sizeof(pthread_t));
16     pthread_create((pthread_t*)self->v_handle, NULL, funcPtr, v_self);
17 }
18
19 " Override this to implement your thread's functionality "
20 Thread run []
21
22 " This is started in a new Pthread "
23 Thread startHelper {
24     /* This function is called by pthread_create(), who will only supply
25         the first parameter called v__closure. However, the value of this
26         parameter has nothing to do with a closure, but is merely
27         (ab)used to transport the self pointer from the parent thread
28         into this new thread. */
29     oop self = ((oop)v__closure);
30     oop selector = _selector("run");
31     _send(selector, self);
32 }
```

Some other common multithreading functionality, such as join and sleep functions, were added to the `Thread` class object as well, but as these were straightforward to implement, there is no need to cover this functionality in detail. Such functions could be implemented by simply making a call to the appropriate function in the Pthreads library.

## 3.2 Synchronization and mutexes

After having provided the `Thread` class object, it already is possible to start, pause and end threads. What should also be added is a mechanism that prevents multiple threads from interfering with each other when they need to work on shared resources. Whereas Java makes use of the `synchronized` keyword to protect such critical sections, C's Pthreads makes use of mutex variables. Since the aj language is closely related to Java, the `synchronized` keyword will also be implemented in the aj language, but this implementation will rely on the Pthreads library's mutexes. Put differently, the semantics of the `synchronized` keyword will now be emulated with mutexes.

A wrapper class object for a Pthread mutex is first created. Implementing this `Mutex` class object was quite trivial, as the only two things that can be done with a mutex are locking and unlocking it. Unlike the `Thread` class object, `Mutex` will not be used as a regular class within the aj language, which is why it has `Object` as its parent object, instead of `ProxifiedObject`. `Object` is a plain prototype-based object and has nothing to do the class-based object-oriented paradigm.

After creating `Mutex`, the PEG of the aj language is adapted to add both the syntax and the semantics for the `synchronized` keyword. The syntax is easily implemented by adding a new type of expression to the `Expr` grammar rule, as shown in appendix A. An example use of the syntax is shown in algorithm 3.3. Unlike Java, there is no need to supply an object that represents a lock. Such a lock object will already be implicitly created for each `synchronized` block. Whilst it is perfectly possible to exactly mimic Java's behaviour, this currently hasn't been done for the sake of simplicity.

**Algorithm 3.3** Using the `synchronized` keyword

```
1 class PrintHelloThread ext Thread {
2     void run(Dummy x) {
3         synchronized {
4             out.println("Hello world")
5         }
6     }
7 }
```

Adding the semantics of the `synchronized` keyword was also relatively easy: When a `synchronized` block opens, create and lock a new mutex;

when the block closes, unlock the mutex. The only pitfall to avoid is not to naively create and lock a new mutex *at runtime* whenever a `synchronized` block opens and then unlock that mutex at the block's end. The word "runtime" forms the source of the problem: Suppose the execution of one thread arrives at a `synchronized` block, a new mutex will be created and locked. Now suppose a second thread also arrives at this very same `synchronized` block. A second mutex is now created and locked. The second thread can execute the block without a problem whilst the first thread also is within the critical section, which obviously isn't the desired effect. This is similar to handing everyone a new key when arriving at a door; anyone could enter and leave without a problem, which defeats the purpose of a key. The solution to this pitfall is to only create one mutex for every occurrence of a `synchronized` block in the source code (, which is somewhat similar to the purpose of a shadow join point, as explained in section 2.1.3). To accomplish this, a dynamic array is created that will hold all mutexes. Whenever the aj parser encounters a `synchronized` block in the source code, a new Mutex is added to the back of the array. Whenever a thread attempts to execute a `synchronized` block, it will fetch the correct mutex from the array and attempt to lock it. The application will use the correct array index because its value was hard coded by the aj parser, which knows which `synchronized` block maps to which mutex in the array.

# CHAPTER 4

# Thread-local aspects

This section represents the main course of this thesis: the addition of thread-local aspects to the aj language. A thread-local aspect is exactly what its name implies: It is an aspect that is local to a thread. This allows different instances of the same thread class to show different behaviour, as one thread could have an aspect applied that isn't present in another thread. The notion of thread-local aspects is quite similar to thread-local storage, where each thread can have its own specific value of the same variable, if this variable is thread-local.

This chapter will introduce and discuss four different variations on the theme of thread-local aspects:

- Correcting the `cflow` pointcut construct, which is inherently associated with thread-local aspects (section 4.2)
- Limiting the scope of an aspect to a thread class (section 4.3)
- Limiting the scope of an aspect to a group of thread instances (section 4.4)
- Creating a `perthread` aspect instantiation strategy (section 4.5)

## 4.1 Applications

Before diving into the what and how of thread-local aspects, it may prove useful to show some motivational examples that demonstrate why one would want to make use of thread-local aspects.

A first example is to use thread-local aspects as a debugging tool[1]. By limiting the scope of an aspect to one particular thread or class of threads, it becomes a useful tool to keep track of what exactly is happening within that thread. Regular debugging tools already have good support for monitoring threads, but these can easily show such an abundance of information that it becomes difficult to find the information that is interesting. Since aspects are written by the developer, they will only focus on the right information. Using aspects also avoids the bad practice of modifying the source code by inserting debugging output statements. A similar example is to use thread-local aspects for profiling purposes, such that the performance of the desired threads can be monitored.

Another purpose of thread-local aspects is in improving an application's performance. Being able to limit the scope of an aspect to a thread enables a finer degree of controlling which join points should belong to the aspect's pointcut. If this means that a pointcut now contains a smaller amount of join points, the aspect's advice is executed less frequently as well, which is likely to result in a performance gain. However, making use of thread-local aspects implies a small amount of overhead, which implies there are some cases where performance is lost.

A last example, possibly the most interesting one, is the application of thread-local aspects in multithreaded autonomic applications. Autonomic systems (20) are also called self-* systems, where the * can be replaced with several terms: self-managing, self-optimizing, self-configuring, self-repairing, ... In other words, an autonomic system can take care of itself. A good example of a potential multi-threaded autonomic application is a web server, where each thread usually represents a user that is visiting the website that is hosted at this server. Thread-local aspects could be used within this context to suit the behaviour of the web server to the user. For example, if the

[1] Using AOP for debugging purposes isn't a new idea(30). Thread-local aspects simply are a refinement.

website has more traffic than it can handle, a thread-local aspect can decide to make its user wait if, for example, the user isn't subscribed to the website, such that more resources become available to other users. Another example is to pay more attention to users that show suspicious behaviour. If such users are present, a thread-local aspect may enable logging just for these users. Whilst it probably already was possible to make a web server adapt itself to its users without using thread-local aspects, it would've proven more difficult to implement. Thread-local aspects are more natural to use in this case, as they have the benefit that a separate state can be kept for each thread, which could contain all sorts of information about the corresponding user.

## 4.2 Adapting the `cflow` pointcut construct

After having read chapter 3, the aj language has minimal support for multithreading, as threads and critical sections can now be created. However, there is one bug that was introduced by adding multithreading support: The `cflow` pointcut construct is broken because it relied on the fact that there only was one thread, being the main thread.

The `cflow` construct puts an additional constraint on the pointcut of an aspect: Only join points within the control flow of a certain method can potentially belong to the pointcut. In other words, if a runtime stack trace were printed at a join point, it must include the specified method in order to comply with the `cflow` construct. An example use of the `cflow` construct is shown in algorithm 4.1, where `SimpleAspect`'s pointcut should only match the call to `inner()` within `test()`. The call to `inner()` within `main()` is ignored.

**Algorithm 4.1** Using the `cflow` construct

```
1  class CflowTest ext ProxifiedObject {
2      void test(Dummy x) {
3          this.inner(x);
4      }
5
6      void inner(Dummy x) {
7          out.println("inner");
8      }
9  }
10
11 class X ext ProxifiedObject {
12     CflowTest s
13
14     void main(X x) {
15         x.s := new CflowTest;
16         x.s.test(0);
17         x.s.inner(0);
18     }
19 }
20
21 aspect SimpleAspect {
22     before call(void Subject.inner(Dummy))
23             && cflow (void Subject.test(Dummy)) {
24         out.println("before");
25     }
26 }
27
28 // === Program output ===
29 //
30 // before
31 // inner
32 // inner
```

The `cflow` construct is currently implemented in aj's PEG by inserting/deploying aspects that use `cflow` as soon as the specified control flow is entered. Such aspects are thus inserted into the right delegation chain dynamically, at runtime. When the specified control flow ends, the aspect is removed/undeployed. The problem that now is introduced with the addition of multiple threads is the fact that there can be multiple control flows at the same time, one for each thread. If one thread enters the control flow specified by `cflow`, the aspect is activated for this thread, but also for all other threads, regardless of whether they're in the specified control flow or not.

In brief, the solution that will be implemented in aj's PEG consists of adjusting the inserted aspect such that it becomes thread-local. The advice is modified such that it will only continue executing itself if the aspect was triggered by the thread that caused the aspect to be deployed. The thread that deployed the aspect must be within the specified control flow. If a second thread happens to enter the specified control flow while the first aspect

is still deployed, a second aspect is deployed for the other thread. If a third thread enters, it also gets its own aspect; and so on...

Implementing this solution is done as follows:

1. First, the code that is executed whenever the specified control flow is entered must be modified. The thread that is currently executing is the one that only is allowed to execute the aspect that is about to be deployed. For this reason, the Pthread handle of the current thread is stored in a variable.

2. The advice of the aspect that will now be deployed is modified as well; this modification happens whilst the application is already running [2]. The advice is modified such that it first checks whether the current thread is the same as the one stored in the previous step. Adding such additional "meta-logic" is also called *residual logic*(15). Residual logic does not contribute to the actual functionality of the aspect, but it acts at a higher level by controlling whether the advice is executed or not.

3. If the current thread equals the one that was stored, the advice can be executed as usual, as the current thread is now guaranteed to be in the control flow specified by the `cflow` construct. If the current thread does not match, the message received by the aspect is resent to its delegate and the remainder of the advice is skipped; from the perspective of the message's sender, the aspect doesn't even exist.

Figure 4.1 demonstrates how delegation chains are modified when a thread-local aspect must be deployed. (The class and the aspect shown in this figure are the same ones as defined in algorithm 4.1.) In step 1, no threads have entered the specified control flow and therefore no aspects are deployed in the delegation chain of the class `CflowTest`. In step 2, thread #1 has entered the control flow and therefore an aspect must be deployed such that it is only executed in this thread #1. In step 3, thread #2 also enters the control flow and a thread-local aspect must be deployed for this thread as well. Suppose thread #2 were to leave the control flow before thread #1, the delegation chain would look just like step 2.

[2] Because the PEGs in COLA make use of a Scheme-like programming language, the powerful quoting functionality is available, which allows the developer to delay the evaluation of an expression to a later point in time.

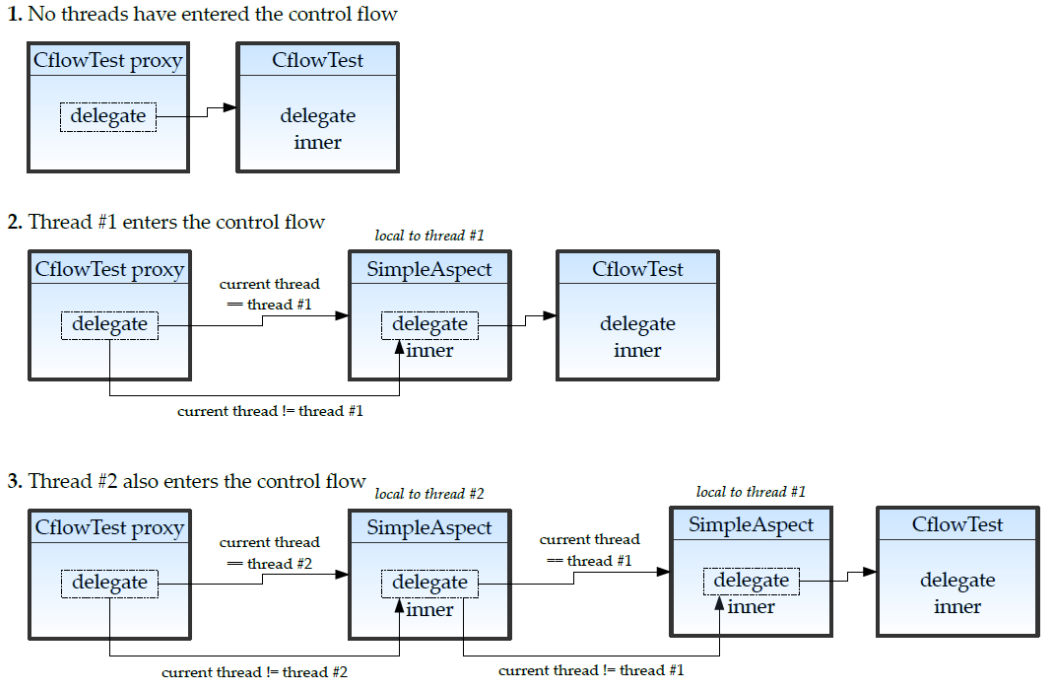


Figure 4.1: Demonstration of the cflow pointcut

Finally, it is important to note that figure 4.1 does not show any indications that the advice of the thread-local aspects is modified. Instead, it seems to imply that the "delegate" message implementation is now performing the check whether the current thread is the right one. (This is indicated by the labels on the arrows.) This wasn't just done because it is easier to clearly show the workings of the `cflow` pointcut in a figure.
Letting the "delegate" message implementation perform the thread-locality check in fact also is a valid alternative solution. It actually makes better use of the fact that the "delegate" message implementation is a function, which is one of the main selling points of delMDSoC's prototype-based model. However, there only is one problem, which also is visible in the figure: The "delegate" message implementation of the delegation chain's proxy object must be altered, which isn't desirable. Section 4.5.4 will solve this problem and even improve this alternative solution, such that it becomes a more viable method than using residual logic.

## 4.3 Aspects local to a thread class

Whereas the `cflow` construct implicitly makes use of thread-local aspects, the pointcut construct introduced in this section explicitly makes use of them. This can be derived from just the name of this pointcut construct: `threadlocal`. It can be used to limit the scope of an aspect to a thread class. Algorithm 4.2 shows an example of how the `threadlocal` pointcut was used to limit the pointcut of `SimpleAspect` to calls to `Printer.print()` within threads of type `PrintHelloThread`.

**Algorithm 4.2** Using the `threadlocal` construct

```
1  class Printer ext ProxifiedObject {
2      void print(String x) {out.println(x);}
3  }
4
5  class PrintHelloThread ext Thread {
6      Printer printer
7
8      void run(Dummy x) {this.printer.print("Hello world");}
9      void setPrinter(Printer x) {this.printer := x;}
10 }
11
12 class PrintByeThread ext Thread {
13     Printer printer
14
15     void run(Dummy x) {this.printer.print("Bye world");}
16     void setPrinter(Printer x) {this.printer := x;}
17 }
18
19 aspect SimpleAspect {
20   after call(void Printer.print(String)) && threadlocal(PrintHelloThread) {
21     out.println("The world replies: Why hello there");
22   }
23 }
24
25 class X ext ProxifiedObject {
26     PrintHelloThread helloThread
27     PrintByeThread byeThread
28     Printer printer
29
30     void main(X x) {
31         x.printer := new Printer;
32
33         x.helloThread := new PrintHelloThread;
34         x.helloThread.setPrinter(x.printer);
35         x.helloThread.start(0);
36         x.helloThread.join(0);
37
38         x.byeThread := new PrintByeThread;
39         x.byeThread.setPrinter(x.printer);
40         x.byeThread.start(0);
41         x.byeThread.join(0);
42     }
43 }
44
45 // === Program output ===
46 //
47 // Hello world
48 // The world replies: Why hello there
49 // Bye world
```

In order to implement the described `threadlocal` construct, it should be possible to map each thread instance to the class it belongs to. To this end a global hash map was created that can map Pthread handles onto the corresponding class name. In the PEG shown in appendix A, this hash map is called the `threadRegistry`. Unfortunately, a thread handle cannot be obtained until the thread is actually started, which means the entries of the

`threadRegistry` can only be created at runtime. Fortunately, this problem can be solved by making use of the `Thread` class's delegation chain.

The solution is to insert an object that intercepts the "run" message in the delegation chain of the thread class specified in `threadlocal`, which executes the thread's behaviour. This object, which actually can be seen as an aspect in its own right, will register the current thread and the corresponding class name in the `threadRegistry`. The aspect will know which class name to use, as its value has been hard coded during the parsing process. Registering the current thread and class name will take place right before executing the thread's run method.

The following question can now be asked: If this registering takes place *before* the thread's behaviour starts running, which thread is executing during the registration? Is it the thread that is being started, or the parent thread that is now spawning the new thread? If it were the latter, the parent thread, the `threadRegistry` would now contain incorrect information. Luckily, at this point in time the new thread is already up and running due to the mechanism that starts new threads, which was discussed in section 3.1. The "startHelper" message implementation in `Thread` is already running inside the new thread; the only responsibility of "startHelper" is to start "run". The object registering the thread in `threadRegistry` is being executed after "startHelper" and before "run"; therefore, the thread that is being registered is the correct one.

The mapping from thread handles to their corresponding class names is now correctly registered in the `threadRegistry`. All that remains is to add thread-local behaviour to all aspects that use the `threadlocal` pointcut construct. This is done in a similar fashion to the previous section, where the advice was modified with residual logic such that only the correct thread can pass through. In this case, the advice of the `threadlocal` aspect is modified such that it will first access the `threadRegistry` to check whether the current thread is of the right class. If so, the current thread is within the scope of the aspect's pointcut and the advice can continue executing as usual. If not, the aspect is skipped.

In summary, implementing the `threadlocal` pointcut consists of two steps. These steps are also illustrated in figure 4.2, in context of the example shown in algorithm 4.2.

1. Intercept the "run" message of the thread class that is specified in the

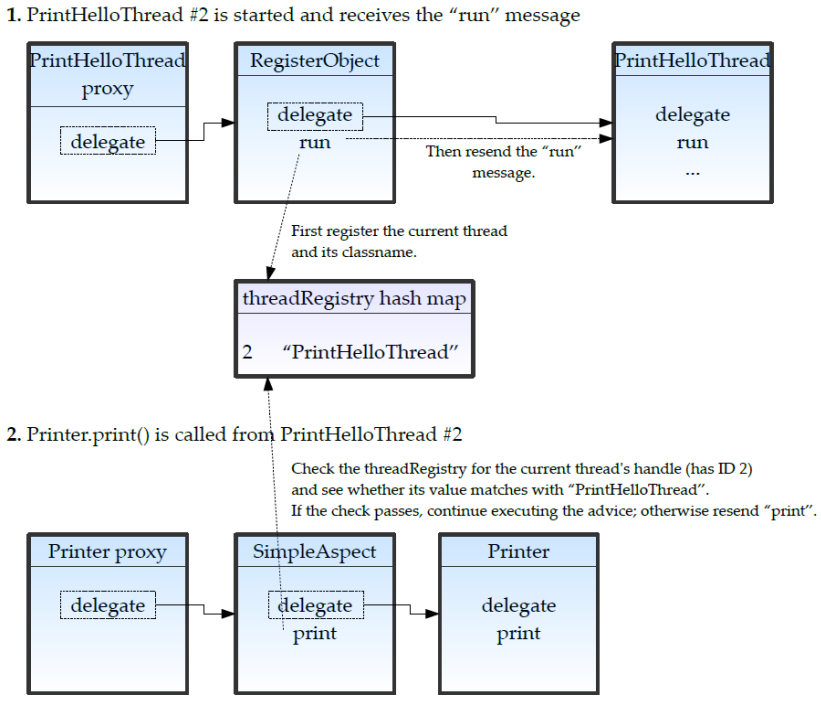


Figure 4.2: Workings of a thread-local aspect that is scoped to a class

`threadlocal` construct. Each time the message is intercepted, the current thread is registered in the `threadRegistry`.

2. Modify the advice such that it will first check the `threadRegistry` whether the current thread is associated with the class specified in the `threadlocal` construct. If this check succeeds, the advice is executed.

The actual implementation of these steps is shown in the `Advice` rule of the aj PEG in appendix A. Whilst the current implementation only supports limiting the scope of an aspect to one class of threads, it should also be quite easy to modify the implementation such that a group of classes can be specified using a regular expression. For example, using `threadlocal(*Server)` in a pointcut would limit the scope of the aspect to all thread classes whose name ends in "Server". The following changes should be made in order to accommodate for multiple classes:

- Instead of only inserting an object into the delegation chain of the class that was specified in a `threadlocal` construct, all thread classes that

match the regular expression will now get such an object. Another possibility is to simply insert such an object into all thread classes. This solution would be easier to implement, but less performant.

- The advice of the aspect would now have to check whether the class corresponding to the current thread matches the regular expression in the `threadlocal` construct.

## 4.4 Aspects local to thread instances

Section 4.3 introduced aspects that can be scoped a thread class, but it may also be useful if it were possible to scope an aspect to specific thread instances. This section will introduce this new flavour of thread-local aspects. A first question that comes to mind is how to specify which instances should be scoped to which thread-local aspect. Instead of improvising even more[3] syntax, Java's annotations make a good candidate as they've become an accepted means to add additional (meta-)information to variables, methods, classes, ... .

[3]Extra syntax on top of an aspect-oriented language should preferably be kept to a minimum, keeping in mind that the syntax of an aspect-oriented language already sits on top of the syntax of an object-oriented language, which can become quite overwhelming.

**Algorithm 4.3** Using annotations and the `threadlocal` construct

```
1  class PrintNumberThread ext Thread {
2      Integer number
3
4      void run(Dummy x) {this.print(0);}
5      void print(Dummy x) {out.println(this.number);}
6      void setNumber(Integer x) {this.number := x;}
7  }
8
9  aspect SimpleAspect {
10   before call(void PrintNumberThread.print(Integer))
11       && threadlocal(@localToSimpleAspect) {
12     out.println("before");
13   }
14 }
15
16 class X ext ProxifiedObject {
17     PrintNumberThread threadOne
18     PrintNumberThread threadTwo
19
20     void main(X x) {
21         x.threadOne := new PrintNumberThread @localToSimpleAspect;
22         x.threadOne.setNumber(123);
23         x.threadOne.start(0);
24         x.threadOne.join(0);
25
26         x.threadTwo := new PrintNumberThread;
27         x.threadTwo.setNumber(456);
28         x.threadTwo.start(0);
29         x.threadTwo.join(0);
30     }
31 }
32
33 // === Program output ===
34 //
35 // before
36 // 123
37 // 456
```

Algorithm 4.3 demonstrates a similar example as in section 4.3, but instead of restricting the aspect to the `PrintHelloThread` class the aspect is now restricted to all thread instances with the annotation `@localToSimpleAspect`. Also note that the `threadlocal` pointcut construct is reused, but instead of passing a class name to the construct it now receives an annotation. The construct can thus be conveniently used for both types of thread-local aspects.

Implementing aspects local to thread instances is very similar to the implementation of aspects local to a thread class. The `threadRegistry` will be used here as well, but next to the hash map that maps thread handles to their class name there now will be a second hash map that will map thread

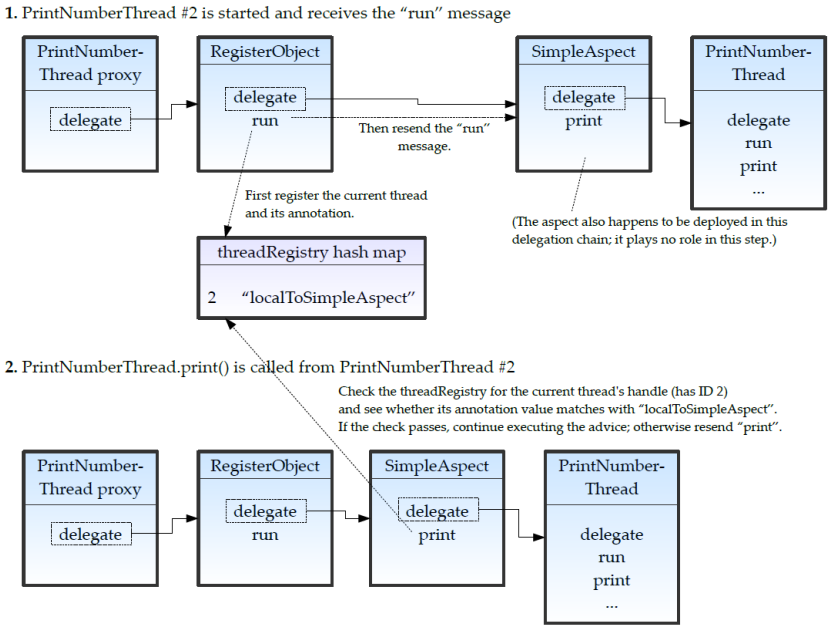


Figure 4.3: Workings of a thread-local aspect that is scoped to instances

handles to their corresponding annotation[4]. (The `threadRegistry`'s first hash map simply won't be used for this kind of thread-local aspects.)

Similar to aspects local to a thread class, the implementation consists of two steps. These two steps are also shown in figure 4.3, in context of the example shown in algorithm 4.3.[5]

1. The delegation chain of all thread instances with an annotation will be modified with an additional object. This object will register the current thread and its corresponding annotation into the `threadRegistry`.

2. The advice of the thread-local aspect is modified. It should first access the `threadRegistry` and see whether the annotation corresponding to the current thread is the same as the specified annotation of the thread-local aspect.

[4]It should be possible to attach multiple annotations to the same instance such that multiple thread-local aspects can refer to the same instances. However, this feature wasn't implemented for the sake of simplicity; the hash map's entries would have to become lists of annotations.

[5]This figure is in fact nearly identical to 4.2; it only looks more complex because, in this example, the aspect happens to be deployed in the delegation chain of the `PrintNumberThread` class, which also contains the object that registers the thread handle.

The implementation of these steps is also shown in the `Advice` rule of the aj PEG in appendix A.

## 4.5 Per thread instantiation

### 4.5.1 Introductory solution

One feature of AOP that hasn't been covered in section 2.1.2 are *aspect instantiation strategies*, also called aspect deployment strategies. These control how aspects are instantiated. As the reader may have already noticed, aspects are never explicitly instantiated in the aj language. Even though this instantiation cannot be seen explicitly, aspects are instantiated just like classes. The default instantiation strategy is "singleton" instantiation, where only one global instance of each aspect is created at the very beginning of the application. Languages such as AspectJ also have several other strategies up their sleeve: "per this" instantiation, "per target" instantiation, "per cflow" instantiation, ... Each has its own purpose. For example, "per this" instantiation creates an aspect instance for each executing object ("this") at join points that were matched by the aspect's pointcut. Because aspects, just like classes, can also have their own state in the form of fields, the state of each aspect instance can now correspond with each executing object.

This section will discuss "per thread" instantiation, where an aspect instance is created for each thread (that contains join points within the aspect's pointcut). While this hasn't been implemented yet in aj's PEG, it is easy to demonstrate its feasibility due to the similarities with the other types of thread-local aspects that were discussed in previous sections. Implementing a similar solution would proceed as follows:

1. Intercept the "run" message of the `Thread` class by inserting a new object into its delegation chain. Each time the message is intercepted, a new aspect instance is inserted into the delegation chain of the target object, as specified by the aspect's pointcut. This results in creating an aspect instance for each thread.

2. When inserting the new aspect, its advice was modified such that only the current thread can pass through. Of course "the current thread"

always is a different thread each time the "run" message was intercepted. This advice modification ensures that each aspect instance is dedicated to only one specific thread.

### 4.5.2 Performance repercussions

Whilst the above solution works, it also is quite naive in terms of performance, as it can tremendously hinder the message lookup cache. The message lookup cache stores the result of the message lookup procedure based on what message was sent and to which object it was sent, the receiver. The given solution creates the problem that, when a message is sent that should eventually end up in one of the thread-local aspect instances, message lookup will always stop its search at the very first of these thread-local aspect instances in the delegation chain. This is the result that will be cached. However, the residual logic of the first thread-local aspect instance will check if the current thread is the one that matches with the thread that is associated with this aspect instance. If not, the message is resent, causing *another* message lookup that will also be cached. Message lookup will immediately end at the object's delegate, which is the next thread-local aspect instance. This process keeps on repeating itself until all thread-local aspect instances are passed.

It is clear that this will have a negative impact on the performance of the application. Not only does it take longer until a message reaches its actual destination; the method lookup cache is spammed with entries that cache the lookup from one thread-local aspect instance to its delegate. This problem aggravates as more and more threads are added to an application. Therefore, this solution clearly isn't meant to scale. Whilst the other kinds of thread-local aspects from sections 4.2, 4.3, 4.4 all work similarly, these won't suffer much from the problem since only one[6] thread-local aspect is introduced in the delegation chain.

### 4.5.3 Modification of the message lookup procedure

An alternative solution that addresses this problem is to adapt the message lookup procedure such that it takes the current thread into account. The procedure will then receive three input parameters: the message that

[6]However, the `cflow` construct can cause multiple aspects to be inserted if multiple threads are in the same specified control flow at the same time.

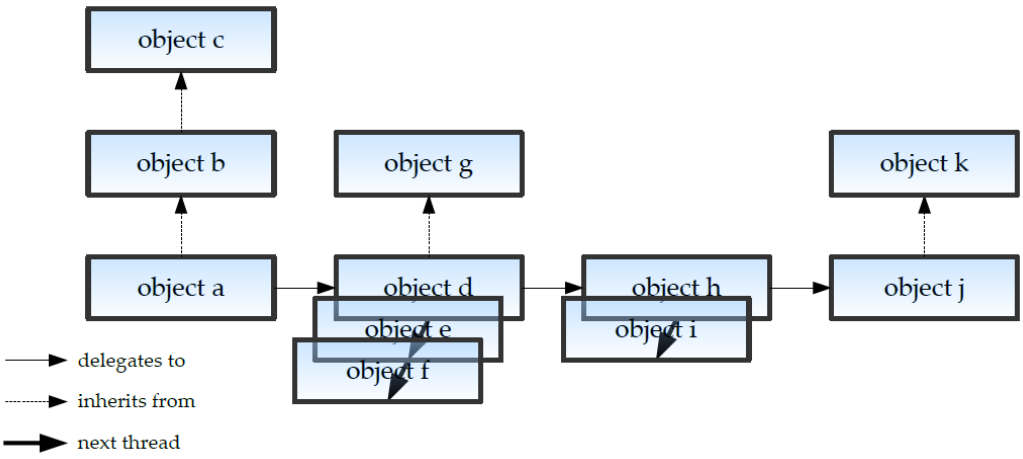


Figure 4.4: Message lookup with the addition of the thread dimension

is sent, the receiver object and the current thread. The procedure that was described in section 2.2.1.1 could now be extended with another dimension, where thread-local objects can be stored. Figure 4.4 shows an example of looking up a message if a third "thread dimension" were present; it is similar to figure 2.1 which demonstrated the procedure with two dimensions. In this example, a message is sent to object a and its implementation is found in object k; the order in which the procedure traverses the objects is: object a, b, c, d, ..., k. In general, the procedure first traverses an object's inheritance chain, then the thread chain and finally its delegation chain, until the appropriate message implementation is found. However, such a procedure is more complicated than it should be. After all, the sequence of letters "a, b, c, d, ... k" is merely one-dimensional. Therefore, the objects can also be rearranged such that the three dimensions are flattened onto just one, leaving only one simple chain of objects.

The benefits of using a message lookup procedure that takes the current thread in account are:

- No more residual logic needs to be added to the advice of an aspect; it can remain unaltered. The lookup procedure now takes care of finding the right thread-local object that matches with the current thread.
- The lookup cache is now used as intended once again. Once a particular lookup operation has been cached, the correct object is directly returned.

However, this approach also has its fair share of disadvantages:

- Just like every object implements the "delegate" message, every object will need to implement an additional message that returns which thread it belongs to (or that it doesn't belong to any thread at all).
- This new lookup procedure will always take into account the current thread, even if the delegation doesn't include any thread-local objects. This affects the overall performance of the application.
- The message lookup cache will now have to map (message, receiver, thread) combinations onto lookup results instead of using the simpler (message, receiver) combinations. This makes reading and writing to the cache more complex and is likely to result in a significantly higher number of cache entries if multiple threads are used.

After considering these advantages and disadvantages, it can be concluded that this solution isn't optimal either, as neither the advantages nor disadvantages seem to be able to outweigh the other. Another lesson learned is that it is best to keep a fundamental mechanism being the message lookup procedure as simple as possible. Because it is almost constantly used, making the procedure more complex is more than likely to have a negative impact on overall performance.

### 4.5.4 Making use of the flexibility of delegates

A last alternative solution is covered in this section. It has little in common with the previous solutions: The message lookup procedure is not modified and no residual logic will be introduced in aspects' advice. This solution will instead rely on the fact that "delegate" message implementations are functions, as already mentioned at the end of section 4.2. These message implementations do not always have to return the same object; they could return a different object based on some meaningful condition.

This solution introduces a "switch object", whose "delegate" message implementation will make use of its dynamic capabilities. A switch object works similar to a railroad switch: If a train arrives at the railroad switch, some sort of control will decide which direction the train will follow. Similarly, if a particular message arrives at a switch object, it can decide to which object out of a set of objects the message is delegated.

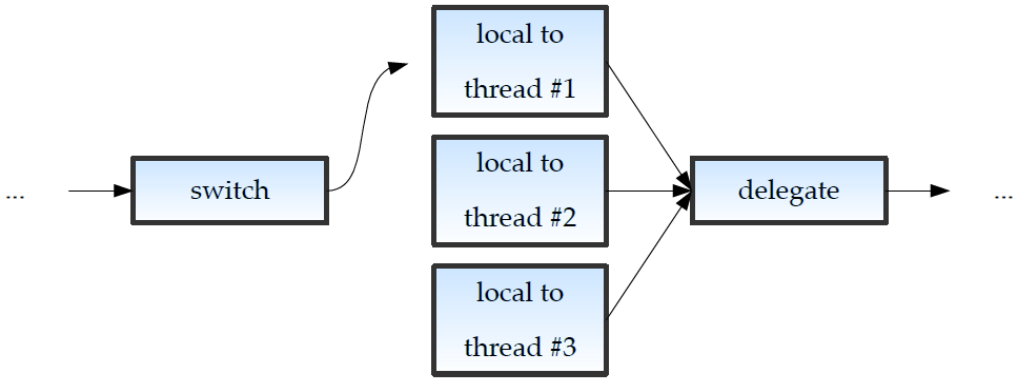


Figure 4.5: Using a switch object for thread-local aspects

Figure 4.5 demonstrates how a switch object can be used to implement per thread instantiation. Whereas the previous two solutions could create a long delegation chain of thread-local aspect instances, the switch object makes use of the fact that only one of these instances is needed at a time. Instead of creating a long delegation chain with all these instances, the switch object's message implementation of "delegate" is such that it always returns the correct thread-local instance. The loose arrow in figure 4.5 depicts that the switch object can choose freely between the available instances, which are labeled "local to thread #...". Depending on the current thread, the "delegate" message implementation of the switch object will return the correct thread-local instance.

The benefits of using a switch object are plentiful:

- No matter how many threads an application has, the length of the possible delegation chains does not grow. The solution therefore is scalable.
- Performance-wise there is little overhead in using per thread instantiation compared to a singleton aspect. The only overhead comes from the fact that the switch object can have multiple delegates; the method lookup cache will have to take this into account.
- As mentioned before, both the message lookup procedure and the advice of aspects remain unmodified. The residual logic that used to be located in advice has now essentially moved to the switch object.
- The prototype-based model does not need any modifications.

The only "disadvantage" is that this solution is only slightly more complex than the two previous solutions.

Finally, the implementation of per thread instantiation using a switch object could proceed in the following steps:

1. The switch object is initialized with two instances of the aspect that is to be deployed. One is for the main thread and the other one serves as a "blueprint" that can be copied whenever a new aspect instance needs to be added because a new thread was started. The reason why the instance for the main thread cannot be used for this purpose is that its state might have changed when a new instance should be created. The switch object can keep track of its instances by, for example, using a hash map that maps Pthread handles onto the corresponding thread-local aspect instances.

2. The switch object is inserted into its target delegation chain.

3. Similar to the solution in section 4.5.1, the "run" message of the `Thread` class is intercepted by inserting a new object into its delegation chain. Whenever the "run" message is received, a message is sent to the switch object that it should create a new thread-local aspect instance for the current thread.

# CHAPTER 5

# Related work

## 5.1 Concurrency models

H. Schippers et al. (24) have recently proposed an actor-based (1) concurrency model that extends delMDSoC's machine model. This extension will allow both true parallelism, which is also provided by this thesis's Pthreads integration, and lightweight concurrency primitives such as coroutines. Unlike Pthreads, the actor model matches more gracefully with delMDSoC's prototype-based model. This immediately becomes apparent with just a quick glance at the concept of the actor model: It treats actors as the primitive construct for concurrent computations. Actors can send and receive messages, make local decisions, create additional actors and determine how to respond to messages. One can quite easily notice the similarities between prototypes and actors. One benefit in favor of this thesis's Pthreads integration is performance related. In Pthreads, state can be shared freely, which implies multiple threads may access the same resource simultaneously. This can lead to performance benefits, but the trade-off is that several problems such as race conditions and deadlocks can arise from this freedom. The actor model avoids these problems by denying direct asynchronous access to

shared state. However, the paper also demonstrates that the actor model extension can be used to map, among others, the threads and locks concurrency model from the Java language onto the machine model of delMDSoC.

Some experiments with multithreading support in COLA had also been conducted, outside the delMDSoC context. In these experiments a read-/write lock was installed in each object, allowing multiple threads to acquire simultaneous read access to an object, but only one object at a time can have write access. Unfortunately, this work hasn't been pursued much further.

## 5.2 Thread-local aspects

As said in chapter 4, thread-local aspects have already been implemented before in other languages and virtual machines, but not yet within the context of delMDSoC. This section will compare the approach taken in this thesis with some of these other languages/VMs.

Steamloom (11, section 3.10.2) is a VM that, similar to delMDSoC, aims to provide virtual machine-level support for AOP. It hasn't entirely reached this goal, as it still expresses AOP mechanisms targeted at an object-oriented machine model. Steamloom has support for thread-local aspects that can be scoped to a single thread. The feature's implementation in Steamloom is also somewhat similar to the implementation presented in this thesis, in the sense that both modify advice and insert residual logic that checks whether the advice should be executed or not. Instead of comparing thread handles, Steamloom gives each aspect an identification number. If a thread is to be associated with a certain aspect, the thread will receive its aspect ID number. The residual logic of an advice can then compare the ID number of its corresponding aspect with the ID number that is stored inside the current thread. This approach requires the thread class to be modified such that it can store aspect ID numbers.

CaesarJ(2) is an AOP language that can limit the scope of an aspect to one thread. Unlike the `threadlocal` pointcut construct in aj, CaesarJ's manner of limiting an aspect is somewhat more implicit. CaesarJ allows the programmer to explicitly instantiate aspect instances, as if they were classes.

One way to activate/deploy the aspect instance is by making use of a `deploy`-block, as shown in algorithm 5.1.

**Algorithm 5.1** CaesarJ's `deploy`-block construct

```
1 MyAspect anAspect;
2 deploy (anAspect) {
3     // Do something useful...
4 }
```

The aspect is only applied to the code within the `deploy`-block; implicitly the aspect is thus limited to the control flow of the code within the block, which also implies it is limited to the thread that is executing this control flow.

AspectWerkz is another AOP language that, as of January 2005, is no longer maintained since its developers have joined forces with the team behind the AspectJ language. What is interesting about AspectWerkz 1.0 specifically is that it has support for per thread instantiation. However, this feature has been removed in the 2.0 version of the language, for a legitimate reason: The language namely also has support for "per cflow instantiation", which can be used to achieve the same effect as per thread instantiation.
Per cflow instantiation can create a new aspect instance for each new control flow amongst the join points matched by a pointcut that is attached to the per cflow instantiation strategy. Per thread instantiation can be simulated by setting the pointcut to calls of the `run` method of the `Thread` class. Each time a new thread starts, a new control flow starts as soon as its run method is executed. The pointcut attached to the cflow instantiation strategy can of course be set to capture other sets of join points as well, which allows for more fine-grained control over when a new aspect instance should be created. Whilst per cflow instantiation hasn't been discussed for the aj language, it shouldn't prove difficult to provide an implementation for this strategy, by essentially combining the knowledge of the adapted cflow pointcut construct from section 4.2 and the per thread instantiation strategy from section 4.5.

Other AOP languages supporting per thread instantiation are JasCo (28), a language tailored for component-based development, and AWED (18), a language that focuses on AOP in a distributed environment. AspectS (13) is an AOP extension to the Squeak Smalltalk language, which can dynamically deploy and undeploy aspects. With some programmatic effort, As-

pectS can support scoping aspects to threads. A final interesting read by É. Tanter (5) compares different crosscutting mechanisms for implementing dynamically-scoped aspects; amongst the group of dynamically-scoped aspects are thread-local aspects.

# CHAPTER 6

# Conclusion and future work

The contributions of this thesis consist of two parts: The first is the integration of the Pthreads library into the delMDSoC virtual machine, such that multiple threads can be created and managed in the aspect-oriented aj language. The second part encompasses four different kinds of thread-local aspects: the `cflow` pointcut construct, aspects local to a thread class, aspects local to a group of thread instances and per thread aspect instantiation. The feasibility of these thread-local aspects within the context of delMDSoC and its prototype-based machine model was demonstrated with implementations for the aj language.

As mentioned in section 2.2, the delMDSoC virtual machine is capable of providing support for a wide array of MDSoC paradigms, of which AOP is one. While this thesis has specifically focused on the aspect-oriented paradigm, the implementation of thread-local behaviour can be carried over to other MDSoC paradigms quite easily. This is due to the fact that the essence of the supporting the several kinds of thread-local aspects in delMDSoC lies in modifying delegation chains by inserting or removing the right objects at the right time. This essence isn't specific to AOP, but is relevant to the underlying prototype-based machine model. All languages built on top of delMDSoC will share this same prototype-based foundation, which is the main reason why thread-local behaviour in other MDSoC languages can be

implemented in a similar fashion as the aspect-oriented aj language. The actual implementation of thread-local behaviour in other languages is left as future work. A good starting language would be the context-oriented (14) cj language, which already is provided in delMDSoC, but does not support thread-local behaviour yet. Whereas aj implicitly instantiates its aspects, cj works with so-called layers, which are explicitly deployed. Despite these differences, providing thread-local constructs will still revolve around dynamically modifying the right delegation chains.

Also left as future work is the implementation of per thread and per cflow instantiation by means of the switch objects discussed in section 4.5.4. These will not only prove useful for thread-local aspects, but may be used for other dimensions as well. For example, a switch object could be useful for switching between several variations of the same piece of functionality, whereas each variation is a point in between the two extremes of a trade-off. This may be useful in autonomic systems, which were also mentioned as an application for thread-local aspects in section 4.1. As a simple example, several different sorting algorithms could be provided, whereas some algorithms may be more suitable for completely random data and others work better for structured data. Each algorithm represents a variation of the same functionality and it's up to the switch object to choose which algorithm would be most appropriate in which cases. It suffices to say that the ability to dynamically change delegation chains is a powerful feature that could have several useful applications next to thread-local behaviour which are worth further investigation.

Using message implementations of "delegate" as a function can also prove to be a useful solution to implement thread-local aspects that are scoped to classes or instances, even though they only introduce one single aspect instance. It wouldn't provide many advantages over using residual logic in the current version of delMDSoC, but in a production-quality virtual machine with a highly optimized message lookup procedure, this would result in a performance boost. However, optimizing the message lookup procedure will prove to be quite the challenge, as it's difficult to cache the lookup result if an object can change around its delegate constantly. The current caching mechanisms do not take this problem into account, which could result in a corrupt cache and an application with incorrect behaviour. As mentioned in section 2.2.2.1, late-binding everything results in a lot of flexibility, but makes it much harder to optimize as well.

# APPENDIX A

## Modified aj parsing expression grammar

This appendix shows the modified version of the aj PEG. One thing to keep in mind is that the semantics of several grammar rules are hidden, as they aren't relevant to either multithreading or thread-local aspects.

Before presenting the aj PEG, a couple of guidelines on how to read a PEG:

- A PEG is similar to the grammars for the yacc parser generator, in the sense that it consists of three different sections: declarations, rules and programs. All three sections make use of the Scheme-like language from COLA's meaning layer.
  - The declarations section, which starts with `%{` and ends with `}%`, is executed at the very beginning, before the aj program is parsed. It can be used for import statements and global declarations.
  - Following the declarations section are the grammar rules that describe the syntax and semantics of a language.
    - A grammar rule has the following syntax:
      `RuleName = Value`
      The `Value` then is a regular expression in terms of all the available rule names.

    - Semantics are added to a rule by inserting curly brackets into the value of a grammar rule.
    - The result of a rule, which is the evaluation of the semantics of that rule, can be assigned to a variable when making use of that grammar rule. This has the following syntax: `varName:RuleName`

  – The programs section ends at the line that contains `%%`. The main purpose of the code in this section is to start the parsing process.

- Anything after a semicolon is ignored and thus treated as a comment. (Comments are also shown in a different font in this appendix.)

- Objects from COLA's representation layer can be accessed within Scheme-like code by making use of square brackets. Everything inside these square brackets is to be interpreted as a Smalltalk-like language. This construction is similar to nesting JavaScript code within the HTML code of a web page, which also results in a mix of two different languages.

**Parsing expression grammar of the aj language:**

```
1  %{
2  ; ***** PEG declarations section *****
3
4  (define atoi    (dlsym "atoi"))
5  (define Integer (import "Integer"))
6  (define IdentityDictionary (import "IdentityDictionary"))
7
8  ; Copy an AST (not fully recursively), replacing the symbol
9  ; at the third position. If the third symbol is actually an
10 ; AST, replace the third symbol there instead (recursive).
11 ; This is used e.g. to replace the temporary "target" symbols
12 ; in (send 'message target ...) or (define-send 'method target ...)
13 ; code at the moment when the actual target becomes available.
14 (define replaceDummySymbol
15     (lambda (code name)
16       (let ((res [Expression withAll: code]))
17         (if [[code at: '2] isExpression]
18             [res at: '2 put: (replaceDummySymbol [code at: '2] name)]
19             [res at: '2 put: name])
20         res)))
21
22 ; Copy an AST, replacing all occurences of a certain symbol by another one
23 ; Generalized version of replaceDummySymbol, in the sense that it is not
24 ; hardcoded to operate at position 3.
25 ; Copying is crucial, since AST's are sometimes reused at different locations
26 ; This is used e.g. to replace all occurences of ___aspectName___ in advice code
```

```
27 ; by the actual aspect name.
28 (define replaceSymbol
29     (lambda (code sym newsym)
30       (let ((res [Expression new: [code size]]))
31         (for (i '0 2 [[code size] - '1])
32              (if [[code at: i] isExpression]
33                  [res at: i put: (replaceSymbol [code at: i] sym newsym)]
34                  (let ()
35                    [res at: i put: [code at: i]]
36                    (if [[res at: i] isString]
37                      (and [[res at: i] = [sym asString]]
38                         [res at: i put: [newsym asString]]))
39                    (and [sym = [res at: i]] [res at: i put: newsym])))))
40         res)))
41
42
43 ; Name of the class of which to invoke the main method after parsing.
44 ; It is passed as a command line argument, and defaults to "X".
45 (define entryclass '"X")
46
47 ; Collection to keep track of all classes in the program
48 (define classes [OrderedCollection new])
49
50 ; Collection of expressions that will be evaluated
51 ; after the above classes collection has been evaluated
52 (define postProcessing [OrderedCollection new])
53
54 ; Flag which should be enabled when parsing advice.
55 ; If we're parsing advice, "this" should be translated to the
56 ; enclosing aspect, not "self"
57 (define parsingAdvice 0)
58
59 ; Store some code here to import each class.
60 ; It can be put e.g. at the beginning of methods.
61 ; Class names aren't visible there otherwise, because they aren't (and can't)
62 ; be stored in the global namespace.
63 (define classDecls [OrderedCollection new])
64
65 ; Import multithreading-related objects
66 (define Mutex (import "Mutex"))
67 (define ThreadRegistry (import "ThreadRegistry"))
68
69 ; Create an array for storing mutexes
70 (define mutexIndex -1)
71 (define mutexes [OrderedCollection new])
72
73 ; Creates a new mutex identification number
74 (define nextMutexIndex (lambda () (set mutexIndex (+ mutexIndex 1))))
75
76 ; Create a ThreadRegistry
77 (define threadRegistry [ThreadRegistry new])
78
79 %}
80
81 ; ***** PEG rules section *****
82
83 ; A program consists of a number of classes and aspects.
84 ; For each one, some "initialization" code is generated.
```

```
85  ; Evaluate the generated code, then launch the "main"
86  ; method of the entry class (see entryclass).
87  Program =
88      (c:Cls { [classes add: c] } | a:Aspect { [classes add: a] })+ {
89          [StdOut println: '"=== Successfully parsed program ==="]
90          (for (i '0 2 [[classes size] - '1])
91              [[classes at: i] eval])
92          (for (i '0 2 [[postProcessing size] - '1])
93              [[postProcessing at: i] eval])
94          (let ((entryclassSym [entryclass asSymbol])
95                ; launchcode consists of sending 'main to the entryclass
96                (launchcode
97                    `(let (,@classDecls)
98                        (send 'main ,entryclassSym 0
99                            (create-instance ,entryclassSym))))) ; sender = 0
100             [StdOut println: '"=== Executing program ==="]
101             [launchcode eval])
102     }
103
104
105 Cls =
106     coll:CLASS name:ClassName ( EXT basename:ClassName )? LCURLY
107     ( m:Meth | f:Field )*
108     ( intr:Introduction )*
109     RCURLY
110
111 Field =
112     TypeName name:Identifier
113
114 Meth =
115     TypeName name:Identifier LPAREN TypeName X RPAREN
116     LCURLY body:ExprList RCURLY
117
118 Introduction =
119     tgclass:ClassName INTRARROW ( m:Meth | f:Field )
120
121 Aspect =
122     coll:ASPECT name:ClassName LCURLY
123     ( !Modifier ( m:Meth | f:Field ) )*
124     ( intr:Introduction { [[coll third] add: intr] } )*
125     {
126         ; We are starting to parse advice now. Set the parsingAdvice flag,
127         ; so that occurences of "this" will be translated to the
128         ; aspect object instead of "self".
129         (set parsingAdvice 1)
130     }
131     ( a:Advice {
132               ; Aspect name is known by now.
133               ; Hence, replace ___aspectName___ occurences in advice body
134               ; by the aspect's name (see Advice and "this")
135               (let ((acopy (replaceSymbol a '___aspectName___ name)))
136                   [[coll fourth] add: acopy])
137          }
138     )*
139     {
140         ; Finished parsing advice, hence reset parsingAdvice flag.
141         (set parsingAdvice 0)
142     }
```

```
143     RCURLY
144
145 Advice =
146     mod:Modifier pc:Pointcut LCURLY body:ExprList RCURLY {
147         (and [mod = 'before]
148           ; For before advice, add a resend statement at the end
149           [body addLast: `(resend ,@[pc at: 'args])])
150         (and [mod = 'after]
151           ; For after advice, do the resend first, and remember the result.
152           [body addFirst: `(set result (resend ,@[pc at: 'args]))]
153           ; Add a statement to return that result at the end.
154           [body addLast: `result])
155         ; In around advice, "proceed" expressions may occur, which imply
156         ; a resend and set the "result" variable.
157         (and [mod = 'around]
158            (let ()
159              ; return "result" variable
160              [body addLast: `result]))
161
162         ; If an advice uses the threadlocal pointcut construct; we'll now
163         ; modify the advice such that it will only execute if it's
164         ; running in the right thread.
165         ; The thread(s) in question will also get an aspect inserted
166         ; that will register the thread's id as soon as it starts.
167
168         ; If the aspect is local to a certain thread class
169         (if [pc includesKey: 'threadLocalClassName]
170             (let()
171                 ; Add an object into the delegation chain of the thread class
172                 ; that will register the thread's handle.
173                 [postProcessing add:
174                     `(let ( (proxy (insert-proxy
175                             (import ,[[pc at: 'threadLocalClassName]
176                             asString]) ())) )
177                         (define-proxy-send 'run proxy ,@[OrderedCollection new]
178                             (let (,@classDecls)
179                                 [threadRegistry registerClassOfCurrentThread:
180                                     ',[pc at: 'threadLocalClassName]]
181                                 (resend)
182                             )
183                         )
184                     )
185                 ]
186                 ; Modify the advice to add a check that verifies
187                 ; whether we're in the right thread.
188                 ; If we're in the right thread, carry on; if not,
189                 ; don't execute the advice.
190                 [body addFirst: `(if (not
191                         [threadRegistry currentThreadIsOfClass:
192                         ',[pc at: 'threadLocalClassName]])
193                     (return (resend ,@[pc at: 'args]))
194                 )]
195             )
196         )
197
198         ; If the aspect is local to specific thread instances
199         ; with a certain annotation
200         (if [pc includesKey: 'threadLocalAnnotation]
```

```
201             (let()
202                 ; Modify the advice to add a check that verifies
203                 ; whether we're in the right thread.
204                 ; If we're in the right thread, carry on; if not,
205                 ; don't execute the advice.
206                 [body addFirst: `(if (not [threadRegistry
207                         currentThreadHasAnnotation:
208                         ',[pc at: 'threadLocalAnnotation]])
209                     (return (resend ,@[pc at: 'args]))
210                 )]
211             )
212         )
213
214         (let ((adviceCode
215             ; Generate code for inserting a proxy in the target object's
216             ; delegation chain.
217             `(let ((proxy (insert-proxy (import ,[[pc at: 'owner]
218                     asString]) ())))
219                 ; The proxy should understand the relevant message, hence
220                 ; install an according method in its vtable.
221                 (define-proxy-send
222                     ',[pc at: 'methodName]
223                     proxy
224                     ,@[pc at: 'args]
225                     (let (,@classDecls
226                           ; Declare result variable, which is used
227                           ; for after and around advice.
228                           (result 0)
229                           ; Declare aspectname, so that it's available
230                           ; in advice bodies. ("this" is translated to it)
231                           ; However, as the actual aspect name is
232                           ; not known here, generate the
233                           ; dummy symbol ___aspectName___, which is
234                           ; replaced in the "Aspect" parsing rule.
235                           (___aspectName___
236                             ; The following Expression (AST) is reused over
237                             ; several advice. Using an unquote inside forces
238                             ; these Expressions to be unique objects.
239                             ; When modifying them (e.g. replaceSymbol), make
240                             ; sure to work on a (recursive!) copy!
241                             (import "___aspectName___")))
242                       ,@body))
243                   proxy)))
244
245             ; In case of cflow, an additional proxy (continuous weaving proxy)
246             ; should be inserted. This proxy will dynamically apply
247             ; (and undo) above "adviceCode" every time the relevant
248             ; control flow is entered.
249             (if [pc includesKey: 'cflowpat] ; cflow
250               (let ((cfp [pc at: 'cflowpat]))
251                 `(let ((prx (insert-proxy (import ,[[cfp at: 'owner]
252                     asString]) ())))
253                   (define-proxy-send
254                     ',[cfp at: 'methodName]
255                     prx
256                     ,@[cfp at: 'args]
257                     (let ((owner (import ,[[pc at: 'owner] asString]))
258                           ; Insert proxy in target object's delegation chain
```

```
259                          (proxy ,adviceCode))
260                         [self removeDelegate: _self]
261                         ; Resend the message, so that the original
262                         ; method will be executed.
263                         (resend ,@[cfp at: 'args])
264                         ; Undo above proxy insertion
265                         [owner removeDelegate: proxy]
266                         [self insertDelegate: _self])))))
267              ; else (i.e. no cflow), just insert the proxy once.
268              adviceCode))
269      }
270
271  Modifier =
272      BEFORE | AFTER | (TypeName AROUND)
273
274  ; A pointcut; we'll first check wether the threadlocal construct is used
275  ; The pointcut grammar rules will not generate any code directly,
276  ; but some useful information is stored in a dictionary,
277  ; which will be used when parsing the corresponding advice.
278  Pointcut =
279      {
280          (set clsName 0)
281          (set ann 0)
282      }
283      pc:CflowPointcut AND THRLOCAL LPAREN
284      (clsName:ClassName | ann:Annotation) RPAREN
285      {
286          (if (!= clsName 0)
287              [pc at: 'threadLocalClassName put: clsName]
288          )
289          (if (!= ann 0)
290              [pc at: 'threadLocalAnnotation put: ann]
291          )
292          pc
293      }
294      | pc2:CflowPointcut { pc2 }
295
296  CflowPointcut =
297      CFLOW LPAREN cpat:MethSigPat RPAREN AND bpc:BasicPointcut {
298            [bpc at: 'cflowpat put: cpat]
299            bpc
300      }
301      | bppc:BasicPointcut AND CFLOW LPAREN cpt:MethSigPat RPAREN {
302            [bppc at: 'cflowpat put: cpt]
303            bppc
304      }
305      | pc:BasicPointcut { pc }
306
307  ; Return some information which is used in Advice:
308  ; - methodName (method name for methods, field name
309  ;   for getters, + ':' for setters)
310  ; - owner (owning class of the join point)
311  ; - args (x for methods, v for setters, none for getters)
312  BasicPointcut =
313      CALL LPAREN pat:MethSigPat RPAREN { pat }
314      | SET LPAREN spat:FieldPat RPAREN {
315            [[spat at: 'args] add: 'v]
316            [spat at: 'methodName put: [[spat at: 'methodName] , ':]]
```

```
317             spat
318     }
319     | GET LPAREN gpat:FieldPat RPAREN { gpat }
320
321 MethSigPat =
322     TypeName owner:TypeName DOT name:Identifier LPAREN TypeName RPAREN {
323         (let ((dict [IdentityDictionary new])
324               (args [OrderedCollection new]))
325           [args add: 'x] ; user should be able to access the parameter
326           [dict at: 'owner put: owner]
327           [dict at: 'methodName put: name]
328           [dict at: 'args put: args]
329           dict)
330     }
331
332 FieldPat =
333     TypeName owner:TypeName DOT name:Identifier {
334         (let ((dict [IdentityDictionary new])
335               (args [OrderedCollection new]))
336           [dict at: 'owner put: owner]
337           ; prefix with '_' in order to use class-wide accessors
338           [dict at: 'methodName put: ['_ , name]]
339           [dict at: 'args put: args]
340           dict)
341     }
342
343 ; Simplification: only use ExprList in method bodies, not e.g. in conditions of
344 ; IF's. That way, we won't end up with nested collections, and the assumption
345 ; that a single expr's tail is a single Expr (not a collection) will be valid
346 ExprList =
347     collexprs:Expr (SEMICOLON eOther:Expr)* SEMICOLON?
348
349 ; Expr is responsible for basic expressions
350 ; ExprTail makes composite expressions.
351 ; Therefore, there are always 2 cases:
352 ; - tail == 0: return basic expression
353 ; - tail != 0: replace ___dummy___ in tail with basic expression
354 Expr =
355     v:Var vt:ExprTail?
356 |   sv:SpecialVar svt:ExprTail?
357 |   { (set cnt 0) (set ann 0) } NEW cn:ClassName cnt:ExprTail? ann:Annotation? {
358         ; Instantiating a new object
359         (if (== cnt 0)
360             (if (== ann 0)
361                 ; If there is no annotation, simply create an instance
362                 `(create-instance ,cn)
363                 ; In case a thread is being instantiated and an annotation is
364                 ; attached to it, insert an object into the new instance's
365                 ; delegation chain that registers its thread handle and its
366                 ; annotation as soon as the run method is called.
367                 (let ( (instCode `(create-instance ,cn)) (modCode
368                     `(let ( (inst ,instCode) (proxy (insert-proxy inst ())) )
369                             (define-proxy-send 'run proxy
370                                     ,@[OrderedCollection new]
371                                 (let (,@classDecls)
372                                     [threadRegistry
373                                         registerAnnotationOfCurrentThread: ',ann]
374                                     (resend)
```

```
                                ))
                                inst
                        )))
                        modCode
                ))
                (replaceDummySymbol cnt `(create-instance ,cn))
            )
        }
|   IF LPAREN cond:Expr RPAREN LCURLY iftrue:Expr
    RCURLY ELSE LCURLY iffalse:Expr RCURLY
|   SYNC LCURLY body:Expr RCURLY {
        ; Synchronized block
        [mutexes add: [Mutex new]]
        (set mutexIndex (+ mutexIndex 1))
        (let ( (curMutexIndex [Integer value_:(nextMutexIndex)]) )
            `(let ()
                [[mutexes at: ,curMutexIndex] lock]
                ,body
                [[mutexes at: ,curMutexIndex] unlock]
            )
        )
    }
|   str:String strt:ExprTail?
|   num:Number numt:ExprTail?
|   smb:STDOUT smbt:ExprTail?
|   PROCEED LPAREN parg:Expr? RPAREN

; ExprTail parses the tail of a composite expression. Its effects therefore
; apply to the first part of the expression, which was parsed by Expr.
; Hence, a placeholder ___dummy___ symbol is used here, which is replaced in the
; Expr parsing rule when the call to ExprTail returns.
ExprTail =
    DOT fld:Identifier ASSIGN rhs:Expr asgnt:ExprTail?
|   DOT selector:Identifier LPAREN arg:Expr RPAREN t:ExprTail?
|   DOT fieldName:Identifier q:ExprTail?
|   EQUALS cmp:Expr


SpecialVar = (TRUE | FALSE | NULL) ![a-zA-Z0-9]

Var = (THIS | X | R | S | V) ![a-zA-Z0-9]

ClassName =
    - < !BOOL i:Identifier > { i }

TypeName = Identifier ; Including 'bool'

Annotation = AT i:Identifier {i}

Identifier =
    - < !INTRARROW [-A-Za-z_0-9+*/<>]+ ':'? > - {
        [[String value_: (strdup yytext)] asSymbol]
    }

-          = (space | comment)*
space      = ' ' | '\t' | end-of-line
comment    = '//' (!end-of-line .)* end-of-line
end-of-line= '\r\n' | '\n' | '\r'
```

```
433
434 Number    =
435     - < [0-9]+ > { [Integer value_: (atoi yytext)] }
436
437 String    =
438     ; everything allowed except "\\n\r
439     - '"' < [^"\\\n\r]* > '"' { [String value_: (strdup yytext)] } ;"
440
441 CLASS     = - 'class'
442 EXT       = - 'ext'
443 NEW       = - 'new'
444 SYNC      = - 'synchronized'
445 IF        = - 'if'
446 ELSE      = - 'else'
447 LPAREN    = - '('
448 RPAREN    = - ')'
449 LCURLY    = - '{'
450 RCURLY    = - '}'
451 INTRARROW = - '<--'
452 AT        = - '@'
453 DOT       = - '.'
454 SEMICOLON = - ';'
455 ASSIGN    = - ':='
456 EQUALS    = - '=='
457 ASPECT    = - 'aspect'
458 CALL      = - 'call'
459 SET       = - 'set'
460 GET       = - 'get'
461 THRLOCAL  = - 'threadlocal'
462 CFLOW     = - 'cflow'
463 AND       = - '&&'
464 PROCEED   = - 'proceed'
465 BEFORE    = - 'before'  { 'before }
466 AFTER     = - 'after'   { 'after }
467 AROUND    = - 'around'  { 'around }
468 THIS      = - 'this'    {
469                           (if (== 1 parsingAdvice)
470                             '___aspectName___
471                             'self)
472                         }
473 X         = - 'x'       { 'x }
474 S         = - 's'       { 'sender }
475 R         = - 'r'       { 'self }
476 V         = - 'v'       { 'v } ; v is argument value
477 TRUE      = - 'true'    { '1 }
478 FALSE     = - 'false'   { '0 }
479 NULL      = - 'null'    { '0 }
480 BOOL      = - 'bool'
481 STDOUT    = - 'out'  { 'StdOut }
482
483 %%
484 ; ***** PEG programs section *****
485
486 ; Import the delMDSoC kernel such that macros like
487 ; define-class, create-instance, insert-proxy, etc.
488 ; become available to the generated code.
489 (require 'delmdsoc)
490
```

```
491 ; Simpler than define-class, because fields are static and
492 ; an aspect does not need a blueprint.
493 (syntax define-aspect ; name (slots...)
494   (lambda (form compiler)
495     (or (and (== '3 [form size])
496        [[form at: '1] isSymbol]
497        [[form at: '2] isArray])
498   [compiler errorSyntax: form])
499     (let ((aspName [form at: '1])
500     (slotNames [form at: '2])
501     (baseSize  [ProxifiedObject _sizeof]))
502       `(define ,aspName
503   (let ((asp [ProxifiedObject _delegated]))  ; This is the new aspect
504     [[asp _vtable] methodAt: '_sizeof put:
505         (lambda (_closure _self self) ,[SmallInteger value_: baseSize]) with: 0]
506     [[asp _vtable] methodAt: '_debugName put:
507         (lambda (_closure _self self) ,[aspName asString])     with: 0]
508     (export ,[aspName asString] asp) ; export new type to Pepsi global namespace
509     ,@(make-slot-functions 'asp slotNames) ; SlotNames must be prefixed with _
510     asp)))))
511
512 (define FileStream (import "FileStream"))
513
514 ; Start the aj parser using either an input file or input from the console
515 (if [[OS arguments] notEmpty]
516     (let ()
517         (set entryclass [[OS arguments] removeLast])
518         (let ((myyy (yy-new [FileStream on:
519                 [File open: [[OS arguments] removeLast]]])))
520             (or
521                 (yy-parse myyy)
522                 [StdOut println: '"Parsing Failed: SYNTAX ERROR"])))
523     (let ((resulti '0)) ; else
524         (while resulti
525             (let ((myyy (yy-new [[StdIn readStream] prompt: '"j> "])))
526                 (set resulti (yy-parse myyy))))))
```

# APPENDIX B

## Glossary of acronyms

**AOP** Aspect-oriented programming (see section 2.1.2)

**COLA** Combined object-lambda architecture (see section 2.2.2.1)

**delMDSoC** delegation-based MDSoC (see section 2.2)

**MDSoC** Multi-dimensional separation of concerns (see section 2.2)

**JIT** Just-in-time (see footnote 9 on page 15)

**OOP** Object-oriented programming

**PEG** Parsing expression grammar (see section 2.2.2.3)

**SoC** Separation of concerns (see section 2.1.1)

**VM** Virtual Machine

# Bibliography


[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. *An Overview of CaesarJ*, pages 135–173. 2006.

[3] Pascal Costanza. A short overview of aspectl. *In European Interactive Workshop on Aspects in Software (EIWAS'04)*, 23:23–24, 2004.

[4] Winton Davies, Peter Edwards, and Ab Ue. Agent-k: An integration of aop and kqml. 1994.

[5] Éric Tanter. On dynamically-scoped crosscutting mechanisms. *SIGPLAN Not.*, 42:27–33, 2007.

[6] Bryan Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37:36–47, 2002.

[7] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. pages 111–122, Venice, Italy, 2004. ACM.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, May 1983.

[10] Chris Hanson. Mit scheme reference manual. 1995.

[11] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. Vdm Verlag Dr. Mũller, April 2008.

[12] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. page 501–524. Springer, 2007.

[13] Robert Hirschfeld. Aspects - aop with squeak. *In OOPSLA'01 Workshop on Advanced Separation of Concerns, Tampa FL*, 13, 2001.

[14] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, ETH Zũrich*, 7:125–151, 2008.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP*, 1997.

[17] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3:184–195, 1960.

[18] Luis Daniel Benavides Navarro, Mario S udholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. pages 51–62, Bonn, Germany, 2006. ACM.

[19] Description Of and Frank Mueller. Implementing posix threads under unix. *In Proceedings of the Second Software Engineering Research Forum*, pages 253–261, 1992.

[20] Manish Parashar and Salim Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, 3566:247–259, 2005.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, 1972.

[22] Ian Piumarta. Accessible language-based environments of recursive theories. Viewpoints Research Institute, 2006.

[23] Ian Piumarta. Efficient sideways composition in colas via 'lieberman' prototypes. Viewpoints Research Institute, 2007.

[24] Hans Schippers, Tom Van Cutsem, Stefan Marr, and Michael Haupt. Towards an actor-based concurrent machine model. *ECOOP '09, 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2009.

[25] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1944–1951, Honolulu, Hawaii, 2009. ACM.

[26] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. *SIGPLAN Notices*, 43:525–542, 2008.

[27] Mark Charles Skipper. *Formal Models for Aspect-Oriented Software Development*. PhD thesis, Department of Computing, Imperial College London, December 2003.

[28] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. pages 21–29, Boston, Massachusetts, 2003. ACM.

[29] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. page 107–119. ACM, 1999.

[30] Yoshiyuki Usui and Shigeru Chiba. Bugdel: An aspect-oriented debugging system. *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005), 0:790–795*, pages 790–795, 2005.

[31] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26:2004, 2002.