Fine-grained Join Point Model in Compose⋆

A thesis submitted for the degree
of Master of Science
at the University of Twente

Olaf Conradi

August 29th, 2006

Research Group

Twente Research and Education
on Software Engineering
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
Enschede, The Netherlands

Graduation Committee

prof. dr. ir. M. Akşit
dr. ir. L. M. J. Bergmans
G. Güleşir, M. Sc.
ir. P. E. A. Dürr

TRESE
Twente Research & Education
on Software Engineering

University of Twente
Enschede - The Netherlands

Abstract

The goal of software engineering is to solve a problem by implementing a software system. A recurring theme in software engineering is one of modularization: separation and localization of *concerns*, where concerns are all things that are of interest in solving the problem.

Object-oriented software development suffers from the tyranny of the dominant decomposition. Aspect-oriented programming (AOP) is an approach that solves that problem of *crosscutting* concerns. A crosscutting concern has its implementation scattered throughout the program. Crosscutting usually results in code tangling, where the code for concerns becomes intermixed.

The Compose⋆ project implements an AOP language based on the composition filters model. Using Compose⋆, it is possible to define concerns that superimpose filter modules on existing classes in the base program. The granularity of the join point model of Compose⋆ is the interface of objects: filter modules can intercept and modify message calls, thus enabling concerns to change the behavior of the program.

AOP shows improvements in modularity, despite some shortcomings in expressiveness. Some of the shortcomings are the *arranged patterns* problem and the problem of *jumping aspects*. The arranged patterns problem is caused by tight coupling between a concern and the program. An example is defining a crosscutting concern by enumerating the join points by name or according to certain naming conventions. This tight coupling harms evolvability of the program and the reusability of the concern. Jumping aspects is a crosscutting phenomenon which occurs because code needs to be added to components depending on their usage context. An example is enabling or disabling a concern depending on the control flow at run-time. The problem with manually coding an applicability control-mechanism into concern code, is that one is patching the weak expressiveness of the join point model.

To solve these shortcomings in this thesis, a fine-grained join point model is proposed which represents join points by *events* emitted during program execution. A *pattern language* is presented to match events against predefined patterns. This allows for context-sensitive crosscutting beyond the level of object interfaces.

Events are monitored at run-time by a local monitor which can receive events from its associated object, therefore encapsulation of objects is respected. Events are checked for matches using finite state automata and by remembering state in between method calls to an object. If a pattern matches, a set of event filters is evaluated to determine which advice to invoke. A constraint model is proposed,

not only to resolve conflicts at shared join points, but also to prevent conflicts depending the result of earlier invoked concerns.

A design is presented to integrate the fine-grained join point model, its pattern language, and run-time monitoring in the Compose⋆ framework. Some bottlenecks are identified which need to be resolved for a successful implementation.

Acknowledgments

Writing a thesis is no small task. Luckily, a number of people supported me during the process, for which I am grateful.

I would like to thank my supervisor Lodewijk Bergmans for his guidance in Compose⋆ and aspect-oriented programming in general. And, of course, to my graduation committee members Gürcan Güleşir, Pascal Dürr, and István Nagy, for their remarks, insights and valuable input. In addition, my thanks go out to the members of the formal methods group for some interesting discussions we had.

I am grateful to professor Mehmet Akşit. It was his course in design patterns that got me interested in joining the software engineering group for my master's project.

I wish to thank my girlfriend for her patience and support, even in hard times. Also, my parents for making my studies possible and believing in me; and my sister for her encouragements. Without them, none of this work would have been possible.

Finally, thanks to the students of the software engineering and formal methods lab for their comments and making my graduation period quite enjoyable.

Olaf Conradi
olaf@conradi.name
August 29th, 2006
Enschede, The Netherlands

Contents

Figures

Tables

Listings

Nomenclature

ANTLR	ANother Tool for Language Recognition
AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CF	Composition Filters
CIL	Common Intermediate Language
CST	Concrete Syntax Tree
DFA	Deterministic finite automaton
EAOP	Event-based Aspect-Oriented Programming
EBNF	Extended Backus-Naur Form
ERE	Extended Regular Expression
ETL	Extended Temporal Logic
IL	Intermediate Language
JPM	Join Point Model
LTL	Linear Temporal Logic
NFA	Nondeterministic Finite Automaton
OOP	Object-Oriented Programming
PDA	Push Down Automaton
STT	State Transition Table
UML	Unified Modeling Language
VM	Virtual Machine

CHAPTER 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [Holljen, 2004; Dürr, 2004; Vinkes, 2004; Bosman, 2004; Staijen, 2005; Havinga, 2005; Boschman, 2006] at the University of Twente. The chapters have been rewritten for use in the following theses [van Oudheusden, 2006; Conradi, 2006; te Winkel, 2006; Huttenhuis, 2006; Doornenbal, 2006; Huisman, 2006; Spenkelink, 2006]. They serve as a general introduction into Aspect-Oriented Software Development and Compose⋆ in particular.

1.1 Introduction

The goal of software engineering is to solve a problem by implementing a software system. The things of interest are called concerns. They exist at every level of the engineering process. A recurrent theme in engineering is that of modularization: separation and localization of concerns. The goal of modularization is to create maintainable and reusable software. A programming language is used to implement concerns.

Fifteen years ago the dominant programming language paradigm was procedural programming. This paradigm is characterized by the use of statements that update state variables. Examples are Algol-like languages such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [Watt, 1990]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

Functional languages try to solve problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output

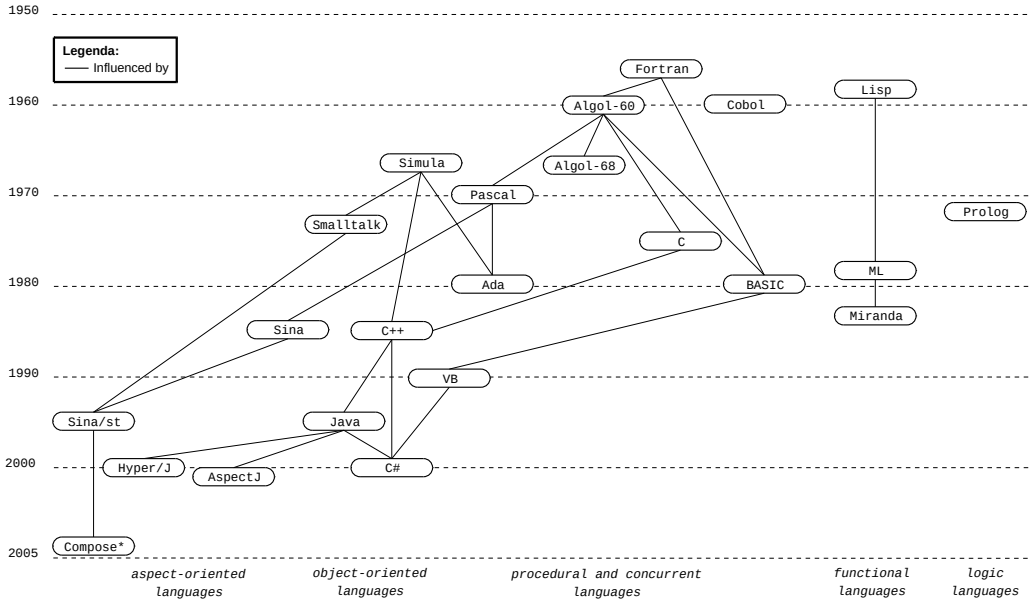


Figure 1.1: Dates and ancestry of several important languages

values from input values. Prolog is currently the most used logic language [Watt, 1990].

A shortcoming of procedural programming is that global variables can potentially be accessed and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

The Object-Oriented Programming (OOP) paradigm improves modularity by encapsulating data with methods inside objects. The data may only be accessed indirectly, by calling the associated methods. Although the concept appeared in the seventies, it took twenty years to become popular [Watt, 1990]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [Gamma et al., 1995].

Existing modularization mechanisms typically support only a small set of decompositions and usually only a single dominant modularization at a time. This is known as the *tyranny of the dominant decomposition* [Tarr et al., 2005]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem. AOP is commonly used in combination with OOP but can be applied to other paradigms as well. The following sections introduce an example to demonstrate

```java
1  public class Add extends Calculation{
2
3    private int result;
4    private CalcDisplay calcDisplay;
5    private Tracer trace;
6
7    Add() {
8      result = 0;
9      calcDisplay = new CalcDisplay();
10     trace = new Tracer();
11   }
12
13   public void execute(int a, int b) {
14     trace.write("void Add.execute(int
         ,int)");
15     result = a + b;
16     calcDisplay.update(result);
17   }
18
19   public int getLastResult() {
20     trace.write("int Add.
         getLastResult()");
21     return result;
22   }
23 }
```

(a) Addition

```java
1  public class CalcDisplay {
2    private Tracer trace;
3
4    public CalcDisplay() {
5      trace = new Tracer();
6    }
7
8    public void update(int value){
9      trace.write("void CalcDisplay.
         update(int)");
10     System.out.println("Printing new
         value of calculation: " +
         value);
11   }
12 }
```

(b) CalcDisplay

Listing 1.1: Modeling addition, display, and logging without using aspects

the problems that may arise with OOP and show how AOP can solve this. Finally, we look at three particular AOP methodologies in more detail.

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented by a method called `write`. Three concerns can be recognized: addition, display, and tracing. The implementation might look something like Listing 1.1. From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering*.

The addition and display concerns are implemented in classes `Add` and `CalcDisplay`, respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 14, and 20 in Listing 1.1a and 2, 5, and 9 in Listing 1.1b). If a concern is implemented across several classes it is said to be scattered. In the example of Listing 1.1 the tracing concern is scattered.

Usually a scattered concern involves code *replication*. That is, the same code is implemented a number of times. In our example the classes `Add` and `CalcDisplay` contain similar tracing code.

In class `Add` the code for the addition and tracing concerns are intermixed. In class `CalcDisplay` the code for the display and tracing concerns are intermixed. If more then one concern is implemented in a single class they are said to be tangled. In our example the addition and tracing concerns are tangled. Also display and tracing concerns are tangled. Crosscutting code has the following consequences:

Code is difficult to change
: Changing a scattered concern requires us to modify the code in several places. Making modifications to a tangled concern class requires checking for side-effects with all existing crosscutting concerns;

Code is harder to reuse
: To reuse an object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

Code is harder to understand
: Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm. Aspect-Oriented Programming (AOP) introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of Aspect-Oriented languages are Sina, AspectJ, Hyper/J, and Compose⋆. A special syntax is used to specify aspects and the way in which they are combined with regular objects. The fundamental goals of AOP are twofold [Gradecki and Lesiecki, 2003]. First, to provide a mechanism to express concerns that crosscut other components. Second, to use this description to allow for the separation of concerns.

Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common join points are method calls. *Pointcuts* describe a set of join points. This allows us to execute behavior at many places in a program by one expression. *Advice* is the behavior executed at a join point.

In the example of Listing 1.2 the class `Add` does not contain any tracing code and only implements the addition concern. Class `CalcDisplay` also does not contain tracing code. In our example the tracing aspect contains all the tracing code. The pointcut `tracedCalls` specifies at which locations tracing code is executed.

The crosscutting concern is explicitly captured in aspects instead of being embedded within the code of other objects. This has several advantages over the previous code.

Aspect code can be changed
: Changing aspect code does not influence other concerns;

```java
public class Add extends Calculation{
  private int result;
  private CalcDisplay calcDisplay;

  Add() {
    result = 0;
    calcDisplay = new CalcDisplay();
  }

  public void execute(int a, int b) {
    result = a + b;
    calcDisplay.update(result);
  }

  public int getLastResult() {
    return result;
  }
}
```

(a) Addition concern

```aspectj
aspect Tracing {
  Tracer trace = new Tracer();

  pointcut tracedCalls():
    call(* (Calculation+).*(..)) ||
    call(* CalcDisplay.*(..));

  before(): tracedCalls() {
    trace.write(thisJoinPoint.
        getSignature().toString());
  }
}
```

(b) Tracing concern

Listing 1.2: Modeling addition, display, and logging with aspects

Aspect code can be reused
: The coupling of aspects is done by defining pointcuts. In theory, this low coupling allows for reuse. In practice reuse is still difficult;

Aspect code is easier to understand
: A concern can be understood independent of other concerns;

Aspect pluggability
: Enabling or disabling concerns becomes possible.

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach every component can be composed with any other component. This approach is followed by Hyper/J, for example. In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. This approach is followed by AspectJ, for example. AspectJ and Hyper/J are covered in more detail in Section 1.4.

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. The three most common approaches to aspect weaving: weaving in source code, weaving in an intermediate language, and weaving in a virtual machine. The first and second approach rely on adding behavior in the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be Intermediate Language (IL) or machine code. Examples of IL are Java byte code and Common Intermediate Language (CIL). The remainder

of this chapter considers only intermediate language targets. The third approach relies on adapting the virtual machine. Each method is explained briefly in the following sections.

1.3.2.1 Source Code Weaving

The source code weaver combines the original source with aspect code. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter the compiler generates an intermediate or machine language output (depending on the type of compiler).

The advantages of using source code weaving are:

High-level source modification
: Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler;

Aspect and original source optimization
: First the aspects are woven into the source code and hereafter compiled by the native compiler. The produced target language has all the benefits of the native compiler optimization passes. However, optimizations specific to exploiting aspect knowledge are not possible;

Native compiler portability
: The native compiler can be replaced by any other compiler as long as it has the same input language. Replacing the compiler with a newer version or another target language can be done with little or no modification to the aspect weaver.

However, the drawbacks of source code weaving are:

Language dependency
: Source code weaving is written explicitly for the syntax of the input language;

Limited expressiveness
: Aspects are limited to the expressive power of the source language. For example, when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

Weaving aspects through an intermediate language gives more control over the executable program and solves some issues as discussed in Section 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that can not be expressed at the source code level. Although IL can be hard to understand, IL weaving has several advantages over source code weaving:

Programming language independence
: All compilers generating the target IL output can be used;

More expressiveness
: It is possible to create IL constructs that are not possible in the original programming language;

Source code independence
: Can add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or run-time
: A special class loader or run-time environment can decide and do dynamic weaving. The aspect weaver adds a run-time environment into the program. How and when aspects can be added to the program depend on the implementation of the run-time environment.

However, IL weaving also has drawbacks that do not exist for source code weaving:

Hard to understand
: Specific knowledge about the IL is needed;

More error-prone
: Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e .g., inlining of methods).

1.3.2.3 Adapting the Virtual Machine

Adapting the virtual machine (VM) removes the need to weave aspects. This technique has the same advantages as intermediate language weaving and can also overcome some of its disadvantages as mentioned in Section 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [Popovici et al., 2002, 2003].

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines
: Using an adapted virtual machine requires that every system should be upgraded to that version;

Virtual machine optimization
: People have spend a lot of time optimizing virtual machines. By modifying the virtual machine these optimizations should be revisited. Reintegrating changes introduced by newer versions of the original virtual machine, might have substantial impact.

1.4 AOP Solutions

As the concept of AOP has been embraced as a useful extension to classic programming, different AOP solutions have been developed. Each solution has one or more implementations to demonstrate how the solution is to be used. As described by Elrad et al. [2001], these differ primarily in:

How aspects are specified
: Each technique uses its own aspect language to describe the concerns;

Composition mechanism
: Each technique provides its own composition mechanisms;

Implementation mechanism
: Whether components are determined statically at compile time or dynamically at run time, the support for verification of compositions, and the type of weaving.

Use of decoupling
: Should the writer of the main code be aware that aspects are applied to his code;

Supported software processes
: The overall process, techniques for reusability, analyzing aspect performance of aspects, is it possible to monitor performance, and is it possible to debug the aspects.

This section will give a short introduction to AspectJ and Hyperspaces, which together with Composition Filters are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [Kiczales et al., 2001] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it, build by several research groups. There are various projects that are porting AspectJ to other languages, resulting in tools such as AspectR and AspectC.

One of the main goals in the design of AspectJ is to make it a compatible extension to Java. AspectJ tries to be compatible in four ways:

Upward compatibility
: All legal Java programs must be legal AspectJ programs;

Platform compatibility
: All legal AspectJ programs must run on standard Java virtual machines;

Tool compatibility
: It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools and design tools;

```
aspect DynamicCrosscuttingExample {
  Log log = new Log();

  pointcut traceMethods():
    execution(edu.utwente.trese.*.*(..));

  before() : traceMethods {
    log.write("Entering " + thisJointPoint.getSignature());
  }

  after() : traceMethods {
    log.write("Exiting " + thisJointPoint.getSignature());
  }
}
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

Programmer compatibility

Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

The points in the execution of a program where the crosscutting behavior is inserted are called *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

The code that should execute at a given join point is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specifies where the additional code is to be inserted. In the example both before and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

Aspects can contain anything permitted in class declarations including definitions of pointcuts, advice and static crosscutting. For example, static crosscutting allows a programmer to add fields and methods to certain classes as shown in Listing 1.4.

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (superclasses and realized interfaces), declare where exceptions need to be thrown, and allow a developer to define the precedence among aspects.

With its variety of possibilities AspectJ can be considered a useful approach for realizing software requirements.

```
aspect StaticCrosscuttingExample {
  private int Log.trace(String traceMsg) {
    Log.write(" --- MARK --- " + traceMsg);
  }
}
```

Listing 1.4: Example of static crosscutting in AspectJ

1.4.2 Hyperspaces Approach

The *Hyperspaces* approach is developed by H. Ossher and P. Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adopts the principle of multi-dimensional separation of concerns [Ossher and Tarr, 2001], which involves:

- Multiple, arbitrary dimensions of concerns;
- Simultaneous separation along these dimensions;
- Ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software life cycle;
- Overlapping and interacting concerns. It is appealing to think of many concerns as independent or orthogonal, but they rarely are in practice.

We explain the Hyperspaces approach by an example written in the *Hyper/J* language. Hyper/J is an implementation of the Hyperspaces approach for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses bytecode weaving on binary Java class files and generates new class files to be used for execution. Although the Hyper/J project seems abandoned and there has not been any update in the code or documentation for a while, we still mention it because the Hyperspaces approach offers a unique AOP solution.

As a first step, developers create hyperspaces by specifying a set of Java class files that contain the code units that populate the hyperspace. To do this is, you create a hyperspace specification, as demonstrated in Listing 1.5.

Hyper/J will automatically create a hyperspace with one dimension—the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units within the specified package. To create a new dimension you can specify concern mappings, which describe how existing units in the hyperspace relate to concerns in that dimension, as demonstrated in Listing 1.6.

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other mappings specify that any method named `trace` or `debug` address the logging and debugging concern, respectively. These later mappings override the first one.

```
Hyperspace Pacman
  class edu.utwente.trese.pacman.*;
```

Listing 1.5: Creation of a hyperspace

```
package edu.utwente.trese.pacman: Feature.Kernel
operation trace: Feature.Logging
operation debug: Feature.Debugging
```

Listing 1.6: Specification of concern mappings

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

Listing 1.7 shows a hypermodule with two concerns, kernel and logging. They are related by a `mergeByName` integration relationship. This means that units in the different concerns correspond if they have the same name (`ByName`) and that these corresponding units are to be combined (`merge`). For example, all members of the corresponding classes are brought together into the composed class. The hypermodule results in a hyperslice that contains all the classes without the debugging feature; thus no `debug` methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. Which makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

Composition Filters is developed by M. Akşit and L. Bergmans at the TRESE group, which is a part of the Department of Computer Science of the University of Twente, The Netherlands. The composition filters (CF) model predates aspect-oriented programming. It started out as an extension to the object-oriented model and evolved into an aspect-oriented model. The current implementation of CF is Compose⋆, which covers .NET, Java, and C.

One of the key elements of CF is the *message*, a message is the interaction between objects, for instance a method call. In object-oriented programming the message is considered an abstract concept. In the implementations of CF it is therefore necessary to reify the message. This *reified message* contains properties, like where it is send to and where it came from.

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, modifying the original flow of the message. To do

```
hypermodule Pacman_Without_Debugging
  hyperslices: Feature.Kernel, Feature.Logging;
  relationships: mergeByName;
end hypermodule;
```

Listing 1.7: Defining a hypermodule

so, a layer called the *interface part* is introduced in the CF model, this layer can have several properties. The interface part can be placed on an object, which behavior needs to be altered, and this object is referred to as *inner*.

There are three key elements in CF: messages, filters, and superimposition. Messages are sent from one object to another, if there is an interface part placed on the receiver, then the message that is sent goes through the input filters. In the filters the message can be manipulated before it reaches the inner part, the message can even be sent to another object. How the message will be handled depends on the filter type. An output filter is similar to an input filter, the only difference is that it manipulates messages that originate from the inner part. The latest addition to CF is superimposition, which is used to specify which interfaces needs to be superimposed on which inner objects.

CHAPTER 2

Compose⋆

Compose⋆ is an implementation of the composition filters approach. There are three target environments: the .NET, Java, and C. This chapter is organized as follows, first the evolution of Composition Filters and its implementations are described, followed by an explanation of the Compose⋆ language and a demonstrating example. In the third section, the Compose⋆ architecture is explained, followed by a description of the features specific to Compose⋆.

2.1 Evolution of Composition Filters

Compose⋆ is the result of many years of research and experimentation. The following time line gives an overview of what has been done in the years before and during the Compose⋆ project.

1985 The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects such as classes from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [Koopmans, 1995].

1987 Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by declarative specifications and the interface predicate construct is added.

1991 The interface predicates are replaced by the dispatch filter, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [Bergmans, 1994].

1995 The Sina language with Composition Filters is implemented using Smalltalk [Koopmans, 1995]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [Glandrup, 1995].

1999 The composition filters language ComposeJ [Wichman, 1999] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.

2001 ConcernJ is implemented as part of a M. Sc. thesis by Salinas [2001]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

2003 The start of the Compose⋆ project, the project is described in further detail in this chapter.

2004 The first release of Compose⋆, based on .NET.

2005 The start of the Java port of Compose⋆.

2006 Porting Compose⋆ to C is started.

2.2 Composition Filters in Compose⋆

A Compose⋆ application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on superimposed objects. Messages have a target, which is an object reference, and a selector, which is a method name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which ob-

```
concern {
  filtermodule {
    internals
    externals
    conditions
    inputfilters
    outputfilters
  }

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 2.1: Abstract concern template

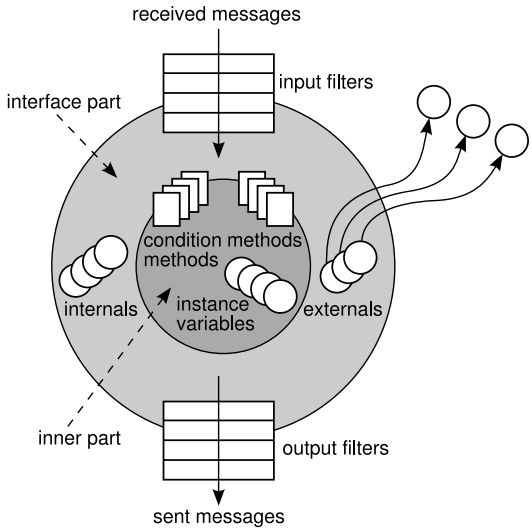


Figure 2.1: Components of the composition filters model

jects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

The working of a filter module is depicted in Figure 2.1. A filter module can contain input and output filters. The difference between these two sets of filters is that the first is used to filter on incoming messages, while the second is used to filter on outgoing messages. The return of a method is not considered an outgoing message. A filter has three parts: a filter identifier, a filter type, and one or more filter elements. A filter element exists out of an optional condition part, a matching part, and a substitution part. These parts are shown below:

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} = \{\overbrace{!pacmanIsEvil}^{condition\ part} =>$$
$$\overbrace{[*.getNextMove]}^{matching\ part}\ \overbrace{stalk\_strategy.getNextMove}^{substitution\ part}\ \}$$

A filter identifier is a unique name for a filter in a filter module. Filters match when both the condition part and the matching part evaluate to true. In the demonstrated filter, every message where the selector is `getNextMove` matches. If an asterisk (*) is used in the target, every target will match. When the condition part and the matching part are true, the message is substituted with the values provided in the substitution part. How these values are substituted, and how the message continues, depends on the type of filter used. At the moment there are four basic filter types defined in Compose★. It is, however, possible to write custom filter types.

Dispatch If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;

Send If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;

Error If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;

Meta If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

The identifier `pacmanIsEvil`, used in the condition part, must be declared in the conditions section of a filter module. Targets that are used in a filter can be declared as internal or external. An internal is an object that is unique for each instance of a filter module, while an external is an object that is shared between filter modules.

Filter modules are superimposed on classes using filter module binding, which specifies a selection of objects on the one side, and a filter module on the other side. The selection is specified in a selector definition. This selector definition uses predicates to select objects, such as `isClassWithNameInList`, `isNamespaceWithName`, and `namespaceHasClass`. In addition to filter modules, it is possible to bind conditions, methods, and annotations to classes using superimposition.

The last part of the concern is the implementation part, which can be used to define the behavior of a concern. For a logging concern, for example, we can define specific log functions and use them as internal.

2.3 Demonstrating Example

To illustrate the Compose⋆ toolset, this section introduces a *Pacman* example. The Pacman game is a classic arcade game in which the user, represented by pacman, moves in a maze to eat vitamins. Meanwhile, a number of ghosts try to catch and eat pacman. There are, however, four mega vitamins in the maze that make pacman evil. In its evil state, pacman can eat ghosts. A simple list of requirements for the Pacman game is briefly discussed here:

- The number of lives taken from pacman when eaten by a ghost;
- A game should end when pacman has no more lives;
- The score of a game should increase when pacman eats a vitamin or a ghost;
- A user should be able to use a keyboard to move pacman around the maze;
- Ghosts should know whether pacman is evil or not;
- Ghosts should know where pacman is located;
- Ghosts should, depending on the state of pacman, hunt or flee from pacman.

2.3.1 Initial Object-Oriented Design

Figure 2.2 shows an initial object-oriented design for the Pacman game. Note that this UML class diagram does not show the trivial accessors. The classes in this diagram are:

Game	This class encapsulates the control flow and controls the state of a game;
Ghost	This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);
GhostView	This class is responsible for painting ghosts;
Glyph	This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;
Keyboard	This class accepts all keyboard input and makes it available to pacman;
Main	This is the entry point of a game;
Pacman	This is a representation of the user controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;
PacmanView	This class is responsible for painting pacman;
RandomStrategy	By using this strategy, ghosts move in random directions;
View	This class is responsible for painting a maze;
World	This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class `Glyph` checks whether movement in the desired direction is possible.

2.3.2 Completing the Pacman Example

The initial object-oriented design, described in the previous section, does not implement all the stated system requirements. The missing requirements are:

- The application does not maintain a score for the user;
- Ghosts move in random directions instead of chasing or fleeing from pacman.

In the next sections, we describe why and how to implement these requirements in the Compose★ language.

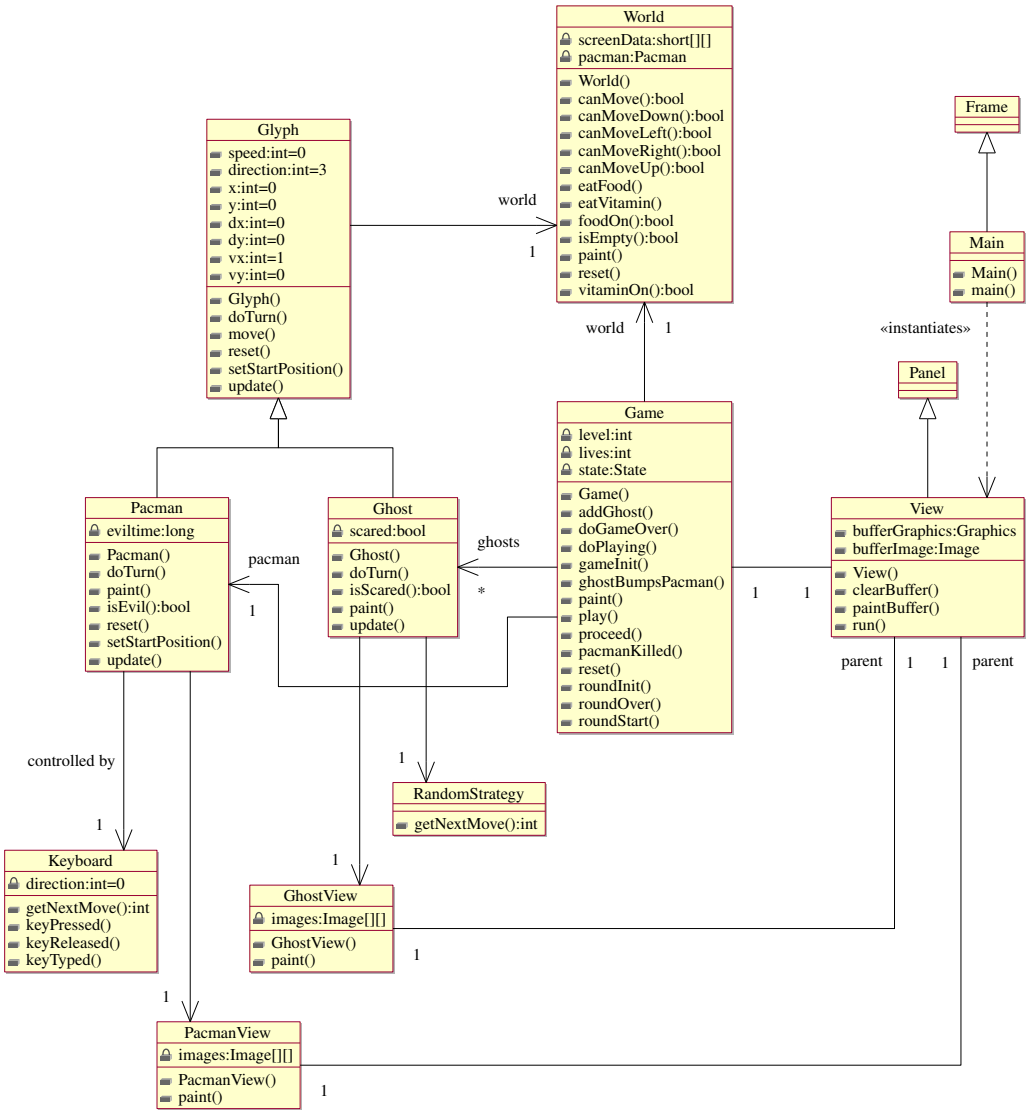


Figure 2.2: Class diagram of the object-oriented Pacman game

```
1  concern DynamicScoring in Pacman {
2    filtermodule dynamicscoring {
3      externals
4        score : pacman.Score = pacman.Score.instance();
5      inputfilters
6        score_filter : Meta = {[*.eatFood] score.eatFood,
7                               [*.eatGhost] score.eatGhost,
8                               [*.eatVitamin] score.eatVitamin,
9                               [*.gameInit] score.initScore,
10                              [*.setForeground] score.setupLabel}
11   }
12   superimposition {
13     selectors
14       scoring = { C | isClassWithNameInList(C, ['pacman.World',
15                                  'pacman.Game', 'pacman.Main']) };
16     filtermodules
17       scoring <- dynamicscoring;
18   }
19 }
```

Listing 2.2: `DynamicScoring` concern in Compose★

2.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pacman game is scoring. This concern involves a number of events. First, the score should be set to zero when a game starts. Second, the score should be updated whenever pacman eats a vitamin, mega vitamin or ghost. And finally, the score itself has to be painted on the maze canvas to relay it back to the user. These events scatter over multiple classes: `Game` (initializing score), `World` (updating score), `Main` (painting score). Thus scoring is an example of a crosscutting concern.

To implement scoring in the Compose★ language, we divide the implementation into two parts. The first part is a Compose★ concern definition stating which filter modules to superimpose. Listing 2.2 shows an example Compose★ concern definition of scoring.

This concern definition is called `DynamicScoring` (line 1) and contains two parts. The first part is the declaration of a filter module called `dynamicscoring` (lines 2–11). This filter module contains one *meta filter* called `score_filter` (line 6). This filter intercepts five relevant calls and sends the message in a reified form to an instance of class `Score`. The final part of the concern definition is the superimposition part (lines 12–18). This part defines that the filter module `dynamicscoring` is to be superimposed on the classes `World`, `Game` and `Main`.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by a class `Score`. Listing 2.3 shows an example implementation of class `Score`. Instances of this class receive the messages sent by `score_filter` and subsequently perform the events related to the scoring concern. In this way, all scoring events are encapsulated in one class and one Compose★ concern definition.

```java
public class Score
{
  private int score = -100;
  private static Score theScore = null;
  private Label label = new java.awt.Label("Score: 0");

  private Score() {}

  public static Score instance() {
    if(theScore == null) {
      theScore = new Score();
    }
    return theScore;
  }

  public void initScore(ReifiedMessage rm) {
    this.score = 0;
    label.setText("Score: "+score);
  }

  public void eatGhost(ReifiedMessage rm) {
    score += 25;
    label.setText("Score: "+score);
  }

  public void eatVitamin(ReifiedMessage rm) {
    score += 15;
    label.setText("Score: "+score);
  }

  public void eatFood(ReifiedMessage rm) {
    score += 5;
    label.setText("Score: "+score);
  }

  public void setupLabel(ReifiedMessage rm) {
    rm.proceed();
    label = new Label("Score: 0");
    label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
    Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo
        .getMessageInfo().getTarget();
    main.add(label,BorderLayout.SOUTH);
  }
}
```

Listing 2.3: Implementation of class `Score`

```
concern DynamicStrategy in Pacman {
  filtermodule dynamicstrategy {
    internals
      stalk_strategy : pacman.Strategies.StalkerStrategy;
      flee_strategy : pacman.Strategies.FleeStrategy;
    conditions
      pacmanIsEvil : pacman.Pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {!pacmanIsEvil =>
                        [*.getNextMove] stalk_strategy.getNextMove};
      flee_filter : Dispatch = {
                        [*.getNextMove] flee_strategy.getNextMove}
  }
  superimposition {
    selectors
      random = { C | isClassWithName(C,
                        'pacman.Strategies.RandomStrategy') };
    filtermodules
      random <- dynamicstrategy;
  }
}
```

Listing 2.4: `DynamicStrategy` concern in Compose⋆

2.3.2.2 Implementation of Dynamic Strategy

The last system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should, depending on the state of pacman, hunt or flee from pacman. We can implement this concern by using the strategy design pattern. However, in this way, we need to modify the existing code. This is not the case when we use Compose⋆ *dispatch filters*. Listing 2.4 demonstrates this.

This concern uses *dispatch* filters to intercept calls to method `RandomStrategy.getNextMove` and redirect them to either `StalkerStrategy.getNextMove` or `FleeStrategy.getNextMove`. If pacman is not evil, the intercepted call matches the first filter, which dispatches the intercepted call to method `StalkerStrategy.getNextMove` (line 9). Otherwise, the intercepted call matches the second filter, which dispatches the intercepted call to method `FleeStrategy.getNextMove` (line 11).

2.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 2.3. The Compose⋆ architecture can be divided in four layers [Nagy, 2006]: IDE, compile-time, adaptation, and run-time.

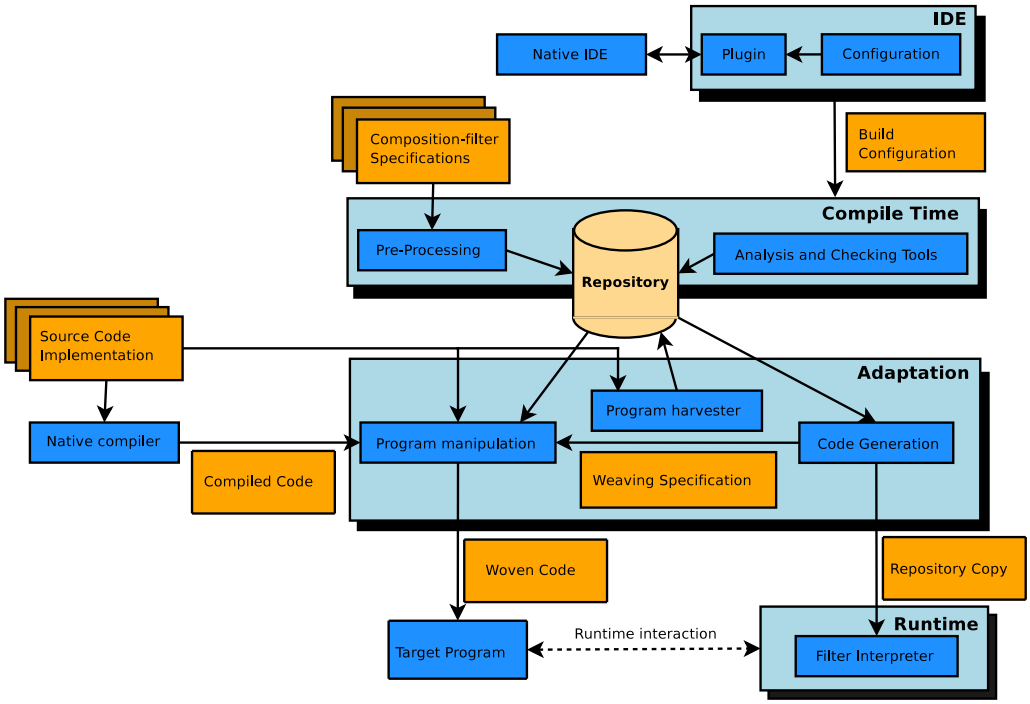


Figure 2.3: Overview of the Compose★ architecture

2.4.1 Integrated Development Environment

One of the purposes of an integrated development environment (IDE) layer is to provide an interface to the native IDE and create a build configuration. A build configuration specifies which source files and settings are required to build an application with Compose★.

A build configuration can be created manually, or by using a plug-in. Examples of these plug-ins are the Visual Studio add-in for Compose★/.NET and the Eclipse plug-in for Compose★/J and Compose★/C. After creating a build configuration, the compile-time is started.

2.4.2 Compile-time

The compile-time layer is platform independent and reasons about the correctness of a composition filter specification with respect to a program. This allows the target program to be build by the adaptation layer.

The compile-time 'pre-processes' the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process, a blackboard architecture is chosen. This means that the compile-time uses a general knowledgebase, called the 'repository'. This knowledgebase contains the structure and metadata of the program. It is used by different modules to base their activities on. Examples of analysis and validation tools are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and val-

idation of the superimposition specification. The compile-time layer creates a weave specification that is used as input by the adaptation layer.

2.4.3 Adaptation

The adaptation layer consists of modules for program manipulation, type harvesting, and code generation. These modules connect the platform independent compile-time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program and adds this information to the knowledgebase. During code generation a reduced copy of the knowledgebase is generated along with a weave specification. This weave specification is used by the weaver during program manipulation to instrument the target program with hooks to the run-time environment of Compose⋆.

2.4.4 Run-time

The Compose⋆ run-time environment is responsible for execution of concerns at join points. It is activated by the hooks present in the program, as woven by the adaptation layer. A reduced copy of the knowledgebase, containing the necessary information for filter evaluation and execution, is used for evaluation of messages. When an instrumented function is called, that message is evaluated by the imposed filter modules. Depending on the condition part and matching part of a filter, accept or reject behavior is executed.

2.5 Supported Platforms

The composition filters concept of Compose⋆ can be applied to any programming language, given that certain assumptions are met. Currently, Compose⋆ supports three platforms: .NET, Java and C. For each platform different tools are used for compilation and weaving. They all share the same platform independent compile-time.

Compose⋆/.NET targets the .NET platform and is the oldest implementation of Compose⋆. Its weaver operates on CIL byte code. Compose⋆/.NET is programming language independent as long as the programming language can be compiled to CIL code. An add-in for Visual Studio is provided for ease of development. Compose⋆/J targets the Java platform and provides a plug-in for integration with Eclipse. Compose⋆/C contains support for the C programming language. The implementation is different from the Java and .NET counterparts, because it does not have a run-time environment. The filter logic is woven directly in the source code. Because the language C is not based on objects, filters are woven on funtions based on membership of sets of functions. Like the Java platform, Compose⋆/C provides a plug-in for Eclipse.

2.6 Features Specific to Compose⋆

The Composition Filters approach uses a restricted (pattern matching) language to define filters. This language makes it possible to reason about the semantics of the concern. Compose⋆ offers three features that use this possibility, which contribute to more control and correctness over the application under construction. These features are:

Ordering of filter modules
: It is possible to specify how the superimposition of filter modules should be ordered. Ordering constraints can be specified in a fixed, conditional, or partial manner. A fixed ordering can be calculated exactly, whereas a conditional ordering is dependent on the result of filter execution and therefore evaluated at run-time. When there are multiple valid orderings of filtermodules on a join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

Filter consistency checking
: When superimposition is applied, Compose⋆ is able to detect if the ordering and conjunction of filters creates a conflict. For example, imagine a set of filters where the first filter only evaluates method `m` and another filter only evaluates methods `a` and `b`. In this case the latter filter is only reached with method `m`; this is consequently rejected and as a result the superimposition may never be executed. There are different scenarios that lead to these kinds of problems, e .g., conditions that exclude each other;

Reason about semantic problems
: When multiple pieces of advice are added to the same join point, Compose⋆ can reason about problems that may occur. An example of such a conflict is the situation where a real-time filter is followed by a wait filter. Because the wait filter can wait indefinitely, the real-time property imposed by the real-time filter may be violated.

The above mentioned conflict analyzers all work on the assumption that the behavior of every filter is well-defined. This is, however, not the case for the user-defined meta filter. Its behavior is unpredictable and therefore poses a problem for the analysis tools.

Furthermore, Compose⋆ is extended with features that enhance usability. These features are briefly described below:

Integrated Development Environment support
: The Compose⋆ implementations all provide a plug-in for an IDE. Compose⋆/.NET provides a plug-in for Visual Studio, while Compose⋆/J and Compose⋆/C provide one for Eclipse;

Debugging support
: The debugger shows the flow of messages through the filters. It is possible to place breakpoints to view the state of the filters;

Incremental building process
: When a project is build and not all the modules are changed, incremental

building saves time.

The following language properties of Compose⋆ can also be seen as features:

Language independent concerns
: A Compose⋆ concern can be used for all the Compose⋆ platforms, because the composition filters approach is language independent;

Reusable concerns
: The concerns are easy to reuse, through the dynamic filter modules and the selector language;

Expressive selector language
: Program elements of an implementation language can be used to select a set of objects to superimpose on;

Support for annotations
: Using the selector, annotations can be woven at program elements. At the moment annotations can be used for superimposition.

CHAPTER 3

Motivation

We will introduce an example to compare traditional object-oriented design with an aspect-oriented design in Compose⋆. The aspect-oriented design shows improvements in modularity, despite some shortcomings in expressiveness.

3.1 Inventory Example

Assume, we want to keep an inventory of all products currently available in stock. A class called `Inventory` will be responsible for the keeping track of all products in stock. It contains methods for adding and removing products. The class `InventoryDisplay` will produce a list of all products currently held in stock. Products are represented by instances of class `Product`. The main entry point for our example is shown in Listing 3.1 and programmed in C#.

To this application we add a notification to produce a new list of products whenever a new product is added or removed. This can be achieved by applying the *observer design pattern* as depicted in Figure 3.1. The observer design pattern defines a one-to-many dependency between objects. When one object changes state, all its dependents are notified and updated automatically [Gamma et al., 1995]. By encapsulating the roles in abstract classes you avoid tight coupling

```
static void Main(string[] args)
{
  Inventory i = new Inventory();
  InventoryDisplay id = new InventoryDisplay(i);

  i.AddProduct(new Product("A"));
  i.AddProduct(new Product("B"));

  id.DisplayInventory();
}
```

Listing 3.1: Main code for creating an inventory containing two products

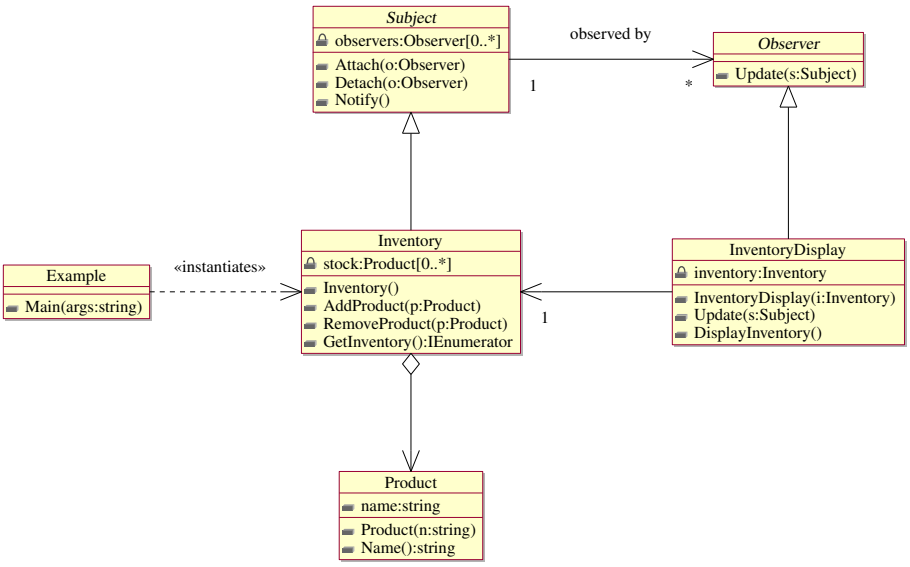


Figure 3.1: Class diagram of inventory application after applying the observer design pattern

between a subject and its observers. Class `Inventory` will play the role of subject and class `InventoryDisplay` will play the role of observer. Our concrete subject `Inventory` is unaware of its observers, all it knows about change propagation is calling method `Notify` whenever necessary. This allows us to remove line 9 from the `main` entry point of our application.

The abstract classes are implemented as illustrated in Listing 3.2. Abstract class `Subject` stores a list of observers and provides an interface for attaching, detaching and notifying observers. Interface `Observer` provides an interface for handling update requests from subjects. These two abstract interfaces are the only coupling between subjects and observers.

This means a concrete subject like `Inventory` should send a notification when its state changes as illustrated by lines 13 and 19 of method `AddProduct` in Listing 3.3a. A concrete observer like `InventoryDisplay` needs to implement an update interface. This is shown in Listing 3.3b. Lines 8 and 11–16 show the necessary changes in applying the observer design pattern.

While the observer pattern does not introduce tight coupling between the classes involved, it does introduce code which does not belong to the core functionality of each class. The base code, classes `Inventory` and `InventoryDisplay`, are not oblivious with respect to the observer code. If the need arises for more complex update handling, further refactoring is necessary. A series of update changes, i.e., adding products in bulk, will cause several consecutive notifications, which may be inefficient.

```
public abstract class Subject
{
  private ArrayList observers;

  public Subject() {
    observers = new ArrayList();
  }

  public void Attach(Observer o) {
    observers.Add(o);
  }

  public void Detach(Observer o) {
    observers.Remove(o);
  }

  public void Notify() {
    foreach (Observer o in observers) {
      o.Update(this);
    }
  }
}
```

(a) Abstract class for registration and notification of observers

```
public interface Observer
{
  public Update(Subject s);
}
```

(b) Interface to receive update notifications from a subject

Listing 3.2: Implementation of abstract classes of observer pattern

```
public class Inventory : Subject
{
  private ArrayList stock;

  public Inventory()
  {
    stock = new ArrayList();
  }

  public void AddProduct(Product p)
  {
    stock.Add(p);
    Notify();
  }

  public void RemoveProduct(Product p)
  {
    stock.Remove(p);
    Notify();
  }

  public IEnumerator GetInventory()
  {
    return stock.GetEnumerator();
  }
}
```

(a) Implementation of `Inventory`

```
public class InventoryDisplay : Observer
{
  private Inventory inventory;

  public InventoryDisplay(Inventory i)
  {
    inventory = i;
    i.Attach(this);
  }

  public void Update(Subject s)
  {
    if (inventory == (Inventory)s) {
      DisplayInventory();
    }
  }

  public void DisplayInventory()
  {
    Console.WriteLine("List of Products
        " + "in Inventory:");
    IEnumerator iter = inventory.
        GetInventory();
    while(iter.MoveNext()) {
      Console.WriteLine(((Product)iter.
          Current).Name);
    }
  }
}
```

(b) Implementation of `InventoryDisplay`

Listing 3.3: Implementation of concrete inventory classes after applying observer pattern

```
concern InventoryObserver {
  filtermodule SubjectInventoryFilter {
    internals
      subject: Subject;
    inputfilters
      attach: Dispatch = {[*.Attach] subject.Attach};
      detach: Dispatch = {[*.Detach] subject.Detach};
      change: Meta = {[*.AddProduct] subject.Notify,
                      [*.RemoveProduct] subject.Notify}
  }

  filtermodule ObserverInventoryDisplayFilter {
    internals
      observer: ObserverInventoryDisplay;
    inputfilters
      register: Meta = {[*.InventoryDisplay] observer.Register};
      update: Dispatch = {[*.Update] observer.Update}
  }

  superimposition {
    selectors
      subjects = {C | isClassWithName(C, 'Inventory')};
      observers = {C | isClassWithName(C, 'InventoryDisplay')};
    filtermodules
      subjects <- SubjectInventoryFilter;
      observers <- ObserverInventoryDisplayFilter;
  }
}
```

Listing 3.4: Observer pattern as concern for inventory example

3.2 Aspect-Oriented Inventory

We can recognize three concerns in our inventory example of Section 3.1. The main concern is keeping track of products in our inventory. A related concern is showing which products are currently available. Our third concern is automatically updating the list of products whenever the inventory changes.

The dominant decomposition into classes allows for the first two concerns to be modular. The update notification crosscuts the first two concerns as shown by the necessary changes in applying the observer pattern. By using aspect-orientation we want to implement the update notification in a modular fashion, and keep the base code oblivious of any concerns.

The abstract observer classes of Listing 3.2 will be encapsulated inside an observer concern and superimposed on the concrete classes as shown in Listing 3.4. We define two filter modules, one for subjects, and one for observers.

The `SubjectInventoryFilter` will dispatch calls for attaching and detaching observers to an internal object (line 6 and 7) which will do the bookkeeping. To notify those observers of a change, `AddProduct` and `RemoveProduct` are intercepted by a meta filter (line 8). After execution of the intercepted message, this meta filter will call `Update` on each attached observer. The implementation of the internal object `Subject` is given in Listing 3.5a.

```
public class Subject
{
  private ArrayList observers;

  public Subject()
  {
    observers = new ArrayList();
  }

  public void Attach(Observer o)
  {
    observers.Add(o);
  }

  public void Detach(Observer o)
  {
    observers.Remove(o);
  }

  public void Notify(ReifiedMessage rm)
  {
    rm.proceed();
    foreach (Observer o in observers) {
      o.Update(this);
    }
  }
}
```

(a) Internal of `SubjectFilter`

```
1  public class ObserverInventoryDisplay :
       Observer
2  {
3    private Subject subject;
4
5    public ObserverInventoryDisplay()
6    {
7      subject = null;
8    }
9
10   public void Register(ReifiedMessage rm)
11   {
12     rm.proceed();
13     subject = (Subject)rm.getArg(0);
14     InventoryDisplay o = (InventoryDisplay
           )rm.getTarget();
15     subject.Attach(o);
16   }
17
18   public void Update(Subject s)
19   {
20     if (subject == s) {
21       ((InventoryDisplay)(object)this).
             DisplayInventory();
22     }
23   }
24 }
```

(b) Internal of `ObserverInventoryDisplayFilter`

Listing 3.5: Implementation of internals for observer concern

The `ObserverInventoryDisplayFilter` will register our concrete observer `InventoryDisplay` with our concrete subject `Inventory` at instantiation time and handle update notifications. The implementation of the internal object `ObserverInventoryDisplay` is given in Listing 3.5b. By intercepting the constructor with a meta filter we will use the information from the reified message for observer registration. The first argument contains the inventory, and the target of the message is our `InventoryDislay` instance (lines 10–16). There is no easy way in Compose⋆ to refer to the inner object from an internal. Methods defined in an internal are woven in the signature of the inner class, but their implementation remains in a separate class. By casting the `this` reference of the internal up to `object` and recasting down to `InventoryDisplay` its public interface becomes available. This allows us to call method `DisplayInventory` on an update (line 21).

Suppose we want to extend our example with a second observer, for example a stock policy which tries to keep the number of products in stock above a certain number. We will need to add a second observer filter module to our observer concern. This filter module, however, will be exactly the same as our display observer, except for the internal object and the name of the constructor. Pa-

```
concern BulkUpdates in ExampleAOP {
  filtermodule BulkUpdate {
    internals
      bulk: ExampleAOP.BulkUpdater;
    inputfilters
      addproducts: Dispatch = {[*.AddProducts] bulk.AddProducts};
      removeproducts: Dispatch = {[*.RemoveProducts] bulk.
          RemoveProducts}
  }

  superimposition {
    selectors
      subjects = {C | isClassWithName(C, 'ExampleAOP.Inventory')};
    filtermodules
      subjects <- BulkUpdate;
  }
}
```

Listing 3.6: Extending inventory with bulk product updates

rameterized filter modules will remove this duplication by passing the name as a parameter. The internal object will need to remain type specific as every observer acts different on updates. To ensure low coupling with the internal of a subject, every observer internal will implement interface `Observer`, whose declaration is the same as given in Listing 3.2b.

If we want to extend our inventory with a method for updating products in bulk, we can add another concern as shown in Listing 3.6. The implementation of our internal for bulk addition of products is shown in Listing 3.7. Removal of products in bulk has a similar implementation which is not shown. We reuse inner method `AddProduct` for the actual addition. Adding products in bulk like this will cause a cascade of update notifications by our observer concern. We can add our new state changing methods to the change filter (line 8 of Listing 3.4). This will add another notification when the bulk update is finished, next to notifications for each single update. This limitation will exist in both object-oriented and aspect-oriented design, unless we add products directly to our stock array (bypassing the inventory interface). For our aspect-oriented design a more expressive pointcut language can circumvent this.

In contrast to the object-oriented design, the aspect-oriented design separates each concern into independent modules. A concern specification binds them together. This keeps coupling low and allows for reusable code.

```
public class BulkUpdater
{
  public BulkUpdater() { }

  public void AddProducts(ArrayList ps)
  {
    foreach (Product p in ps) {
      ((Inventory)(object)this).AddProduct(p);
    }
  }
}
```

Listing 3.7: Internal for adding products in bulk. Removal is not shown, but similar.

3.3 Problem Identification

Like many other patterns, the observer pattern is an often recurring pattern. To improve reuse, filter modules (and their internals) should be independent of application context.

Looking at the `SubjectInventoryFilter` of Listing 3.4, we can see an enumeration of state changing functions. The enumeration needs to be kept up to date whenever we add or remove methods. This coupling between the concern and the superimposed class is known as the *arranged patterns* problem.

By adding bulk updating of products to the inventory example, update notifications became dependent on their method of invocation. When method `AddProduct` of class `Inventory` is called by method `AddProducts`, notification should be deferred until all products are added. This conditional advice execution is known as *jumping aspects*.

3.3.1 Arranged Patterns Problem

An important issue in aspect-oriented languages is describing join points without introducing tight coupling between the aspect the base program. Tight coupling harms evolvability of the program and the reusability of the aspect. Gybels and Brichau [2003] illustrate this as the *arranged patterns* problem.

In early AOP languages crosscuts were formed by explicitly enumerating join points by name. This introduces tight coupling between the aspect and the modules it crosscuts. Most AOP systems today use property-based crosscutting, where name-based enumerative crosscuts are replaced by associating a richer set of properties to each join point. A crosscut tries to express the natural language semantics through conditions on the underlying pattern of each join point. By using wild cards on interface methods, for example. This introduces naming conventions or other ways of structuring code in patterns. These patterns might not be expressive enough and force the programmer to always keep them in mind. To capture all setters of a class, one might use the pattern `set*`, but this could also capture a newly created method called `settlement`, for example. The naming conventions must be followed for the sake of the aspects and not to better express

```
concern InventoryObserver {
  filtermodule SubjectFilter {
    internals
      subject: Subject;
    inputfilters
      attach: Dispatch = {[*.Attach] subject.Attach};
      detach: Dispatch = {[*.Detach] subject.Detach};
      change: Meta = {[@Update *] subject.Notify}
  }
  // ...
}
```

Listing 3.8: Observer concern with annotation based method selection. Methods annotated with `Update` mark state changing methods at the subject.

the intent of the method, this is what Gybels and Brichau call the 'arranged pattern' problem.

A proposed solution to this problem could be the use of annotations to bind design information to a range of language constructs [Nagy, 2006]. By annotating the state changing functions of class `Inventory`, they can be selected in the matching part of a filter, as shown in Listing 3.8. This moves the problem to selecting the functions to annotate and how to add the annotations (automatically based on a query, or by the developer in the source code). In addition, annotations are usually statically bound, while the method might not change state on every invocation, for example. Using an annotation in the matching part of a filter is currently not implemented in Compose⋆, however.

3.3.2 Jumping Aspects

Jumping aspect code is a crosscutting phenomenon, described by Brichau et al. [2000], which occurs because code needs to be added to components depending on their usage context. When components are built to maximize their ease of reuse, it is very likely that they require specific adjustments to use them in a particular application context.

In the inventory example, method `AddProduct` contains a jumping aspect. If called directly, all observers should be notified. If called by method `AddProducts`, only one notify should take place when the operation as a whole is finished. A possible solution to this problem could be the addition of a flag to the aspect code, which conditionally calls `Notify`. The problem with this approach is that we are manually coding an applicability control-mechanism into the aspect code itself. This only patches the weak expressiveness of the aspect language. It should, however, be a task of the weaver. The programmer should only indicate the need for such constructs. "In order to support jumping aspects code we have to reconsider the idea of a crosscut as an interaction between 'locations in the component code' and 'locations from which this component is called" ' [Brichau et al., 2000].

3.4 Problem Statement

The Compose⋆ project offers aspect-oriented software development based on the composition filters model. The current model works at the message level. But sometimes one is required to specify crosscutting behavior beyond the level of object interfaces. To accommodate this, the model will be extended with a selection of statement level events.

A pointcut language will be made to support pattern matching on these events. The solution will try to cooperate with existing composition filters.

CHAPTER 4

Definitions

Most aspect-oriented systems operate within the static structure domain to define crosscutting behavior. This behavior is specified in advice and designated by *pointcuts*. A pointcut describes a set of join points. *Join points* are well-defined places in the structure or execution flow of a program. A join point model (JPM) specifies the kinds of join points allowed.

In this chapter we propose to extend Compose* with a fine-grained join point model in which join points are signalled by events emitted during program execution.

4.1 Definition of Event

The dictionary defines an *event* as a particular occurrence or happening [American Heritage]. This leaves us with a vague notion of things that happen. In physics, an event is usually related to a physical object. The difference between an event and an object is in the mode of being and how they relate in space and time. Objects are said to *exist* and events are said to *occur*. Objects have relatively clear spatial boundaries (location) and unclear temporal boundaries (duration), whereas events have unclear spatial boundaries and clear temporal boundaries. Events take up time and they persist through time by having different stages at different times [Casati and Varzi, 2000]. An event can be distinguished to occur at a point in time because the state of the world changed. Something was different *before* and *after* the event. Events can be of a certain type and the occurrence of an event is said to be an instantiation of such an event type. Events of a certain type have properties to not only signal the occurrence, but also to provide quantitative information regarding that occurrence.

For our purpose, the domain of events is restricted to events occurring during the execution of computer programs. The next section describes the level of quantification for these events, followed by a list of event types.

4.2 Quantification of Events

The expressiveness of a crosscut language depends on the method of quantification. It defines the granularity of reasoning about the program. We can quantify over static structure of a program and over its dynamic behavior. [Filman and Friedman, 2005]. The method of quantification for a language is described in a join point model. Join points can be classified as *structural* and *behavioral* join points. Structural join points are expressed in terms of syntactic constructs of the programming language. For example, classes with certain inheritance relations. Behavioral join points correspond to *events* in the control flow of a program. For example, method invocations or throwing an exception.

There are two types of events, *syntactic* events and *dynamic* events. Syntactic events have a direct mapping to statements in the source code of a program. Dynamic events do not have such a mapping, they depend on the execution flow of the program. They can, however, be bound to certain regions of the code, called the *shadow* of a join point in AspectJ terminology [Hilsdale and Hugunin, 2004]. Examples of syntactic events include method invocation and variable assignment. Examples of dynamic events include run-time exceptions, invoking a method from within the control-flow of another method, thread switching and garbage collection (provided the run-time environment supports it).

A pointcut designator describes a set of join points, where advice should be invoked. It is a quantification mechanism over behavioral or structural join points of a program.

The join point model of Compose* supports selection of structural join points by using superimposition. The selector language supports most structural constructs available in object-oriented languages. Behavioral join points are selected inside filter elements of a filter module. Filter elements are able to select messages at the interface of an object. The set of join points can be refined by certain dynamic conditions, like checking the value of a variable in a filter condition.

To refine the join point model of Compose*, an event-based expression language is defined. The language is used in pointcut definitions and allows for the composition of events inside patterns, along with the ability to filter on the property of events.

This event-based language, not only allows for selection of more types of join points, it also allows for context-sensitive crosscutting. The program is instrumented to generate events at run-time. Event are considered to be atomic, which means they form an ordered sequence. Such a sequence is called a *trace*. Event monitors will observe events and try to match them against a pattern. When a match is found, some advice, as defined along with the pattern, is invoked.

The next section presents an overview of syntactic events.

4.3 Types of Events

Events are the building blocks for creating event patterns. Each event has a set of properties that can be used to filter on specific events. A syntactic event contains both static and dynamic properties, a dynamic event only contains dynamic properties.

There is one dynamic property that is common for all events:

Sender	Current stack frame.

It represents the context in which the event occurred and contains the current stack frame at the time of the event.

4.3.1 Method Events

A method is either defined as a class member, or in non object-oriented languages, as a function. Depending on the programming language, constructors, destructors, and properties (getters and setters of .NET) are a subclass of method events. A difference is made between the call and the execution of a method. A call event is an outgoing message from an object, while an execution event is an incoming message to an object.

The following structural properties can be derived from methods:

Selector	Name of method;
Target class	Class name of method;
Visibility	Public, protected, internal, or private;
Access type	Static or instance;
Parameter names	Array of parameter names;
Parameter types	Array of parameter types;
Return type	Return type of method;
Exception type	Array of types of exceptions the method can throw, not part of the signature in .NET;
Annotations	Annotations of the method, or custom attributes in .NET;
Package	Package or namespace of the target of the message.

In our event model the following dynamic properties are available for method events:

Sender	Sender object reference;
Target	Target object reference;
Parameter values	Array of values of parameters;
Return value	Value of return type;
Exception object	Thrown exception.

Events can be generated before and after a call, before and after an evaluation of every parameter, and finally before and after the actual execution of a call. This is illustrated in Figure 4.1. Related to method events are jump and exception events, e .g., return, as explained in Section 4.3.5.

Further down the chain of events, more dynamic properties will be available as summarized in Table 4.1. The structural properties are already known at compile-time.

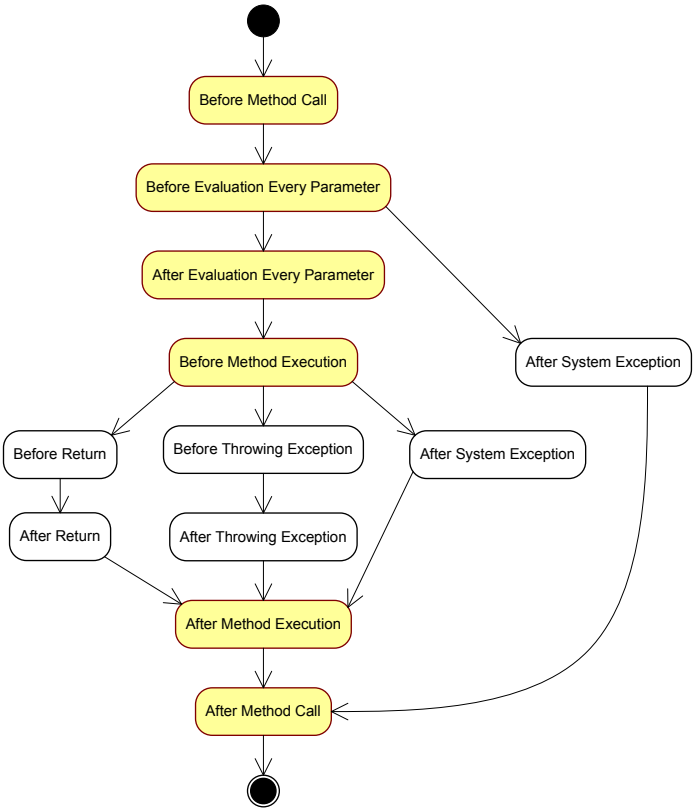


Figure 4.1: UML state diagram containing the flow of events for method invocations. The return and exception events are shown for completeness

Table 4.1: Dynamic properties for method events. The table shows after which event a property is available

Property	**Call**	**Parameters**	**Execution**
Sender object	before	before	before
Target object	before	before	before
Parameter values	after	after	before
Return value	after		after
Exception object	after		after

4.3.2 Data Events

Variables represent storage locations of a certain type. There are different categories of variables: static or instance variables, array elements, parameters (i.e.,, value, reference or output) and global or local scope variables. In the .NET framework global variables don't exist, they can be simulated as part of static classes.

Reference type variables store access to the actual data. In the .NET framework, classes, interfaces, delegates, objects and strings can be referenced. Value type variables always store a value of the associated type (e.g.,, int or float). In the .NET framework each value type has an implicit default constructor and the value, unlike references, cannot contain the `null` value. If code in the .NET framework operates in the "unsafe context", variables are not tracked by the garbage collector. In this context pointer type variables exist and can point to both managed and unmanaged reference types or value types.

If a variable is a data member of a class, it's called a field. Fields are usually declared private. If they need to be accessed by other objects, they can be exposed through properties which will do the modifications in a uniform way. Properties within the .NET framework are implemented through get and set accessors, which are modelled as method calls.

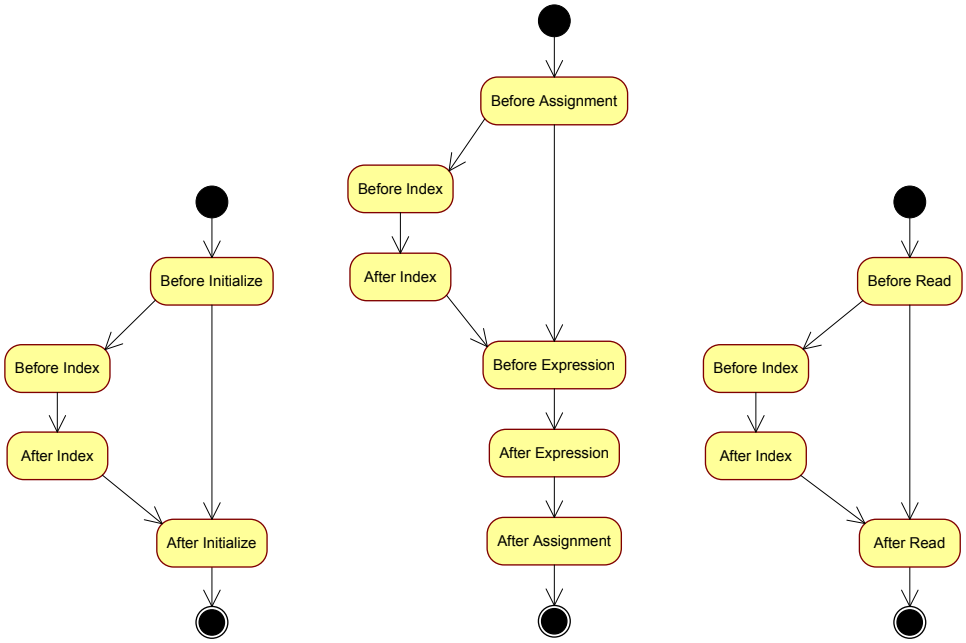
In terms of events, variables can be initialised, assigned, read and disposed of. For variables of the array type, an extra *before* and *after event* is introduced to evaluate the index. The following static properties are available for events dealing with variables:

Variable name	Name of a variable;
Variable type	Type of a variable;
Scope	Local scope, field scope, or parameter scope;
Target class	Name of the class to which a variable belongs (fields only);
Visibility	Public, private or protected (fields only);
Access type	Static or instance (fields only).

The following list shows the dynamic properties of events during manipulation of variables:

Value	Value of variable;
Old value	Old value of variable (in case of an assignment);
Index value	Index value in case of array variable type.

Table 4.2 shows the availability of the dynamic properties. In a *before assignment* event, the value property contains the old value, after assignment it contains the new value. The previous value will still be available through property old value. In the .NET framework and the Java language, automatic garbage collection takes care of disposing variables. The signalling of these events are only possible if the run-time environment makes them available. The flow of events is depicted in Figure 4.2.



(a) Initialise (b) Assignment (c) Read

Figure 4.2: UML state diagram containing the flow of events for the initialisation, assignment and reading of variables. The index evaluation events only exist for array type variables.

Table 4.2: Dynamic properties for variable events. The table shows after which event a property is available.

Property	**Index**	**Initialise**	**Assign**	**Read**	**Dispose**
Value		after	before/after	before	before
Old value			after		
Index value	after	before	before	before	before

Table 4.3: Selection events and when their associated dynamic properties are known.

Property	**Selection**	**Condition**	**Branch**
Condition value	after	after	before
Branch label(s)	after	after	after

4.3.3 Selection Events

Selection statements transfer the control flow of a program to a specific branch based on the outcome of a condition. Most programming languages support two kinds of selection statements: `if` and `switch`.

The `if` statement makes a selection based on the value of a boolean expression. If the expression evaluates to true the first branch is chosen, or, if there is one, the `else` branch. The `switch` statement evaluates an expression and chooses the branch with the corresponding label as the result, or, if there is one, takes the branch labelled `default`. In the .NET framework `goto` statements are allowed to jump to a different branch inside the body of a `switch` statement. A `break` statement always transfers control out of the selection statement. Jump statements shall be discussed in Section 4.3.5.

The only static property both selection statements have, is a list of branch labels.

Branch labels — Branches shows a list of branch labels for the selection statement.

The dynamic properties consist of the value of the condition and the chosen branch. Table 4.3 shows after which event which dynamic property is available.

Condition value — Value of condition;
Branch label — Label of chosen branch or branches (`switch` allows jumping between branches).

In terms of events, both selection statements signal the same sequence of events. There is the before and after event of the `if` or `switch` statement, the before and after evaluation of the condition expression, and the before and after selection of the chosen branch. They are illustrated in Figure 4.3.

4.3.4 Iteration Events

Iteration events repeatedly execute an embedded code block until a certain loop-termination criteria is met. Most languages support `for`, `while` and `do while` statements. The .NET framework also contains a `foreach` statement.

Iteration events do not have any static properties, only dynamic ones:

Condition value — Value of the loop condition;
Iterations — Number of iterations.

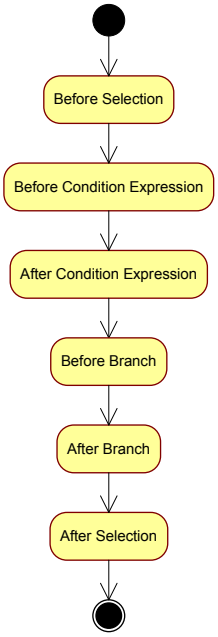


Figure 4.3: UML state diagram containing the flow of events for selection statements, where selection corresponds to either an `if` or `switch` statement

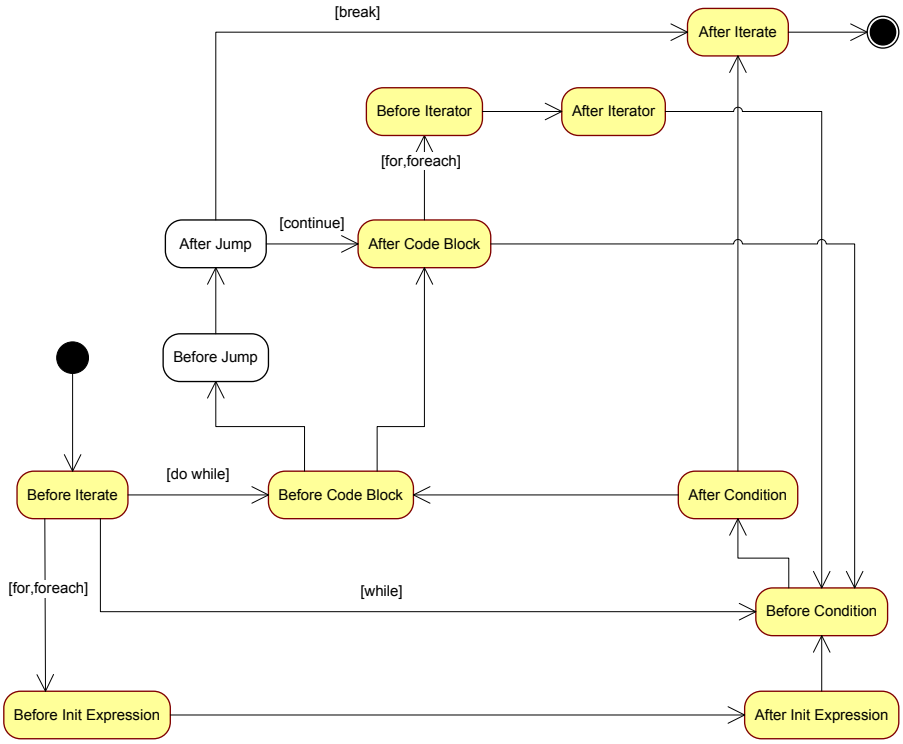


Figure 4.4: UML state diagram containing the flow of events for iteration statements. The guards differentiate between the different types of iteration (`for`, `foreach`, `while` and `do while`). The jump statements `break` and `continue` are not part of the iteration events, they are shown to see how they fit in

Table 4.4: Dynamic properties for iteration events. The table shows after which event the property is available

Property	Iteration	Initialise	Condition	Code block	Iterator
Condition value		after	before (after 1st)	before	before (after 1st)
Iteration count	after	before	before	before	before

Table 4.4 lists the dynamic properties and after which event they are available. In terms of events all iteration statements support the following events: before and after event for the iteration statement, before and after event for the termination condition, and finally before and after event for the embedded code block. The `for` statement also includes the events before and after condition initialisation, and before and after iterators (to update a counter for example). This is illustrated in Figure 4.4.

4.3.5 Jump and Exception Events

Jump statements unconditionally transfer control, usually outside the current code block. Jump statements consist of `break`, `continue`, `goto`, `return` and `throw` statements. When a jump statement occurs inside a `try` block, the control is first transferred to the associated `finally` block(s).

The `return` and `throw` statements have a before and after evaluation of an expression. In case of `return` the expression must be of the same type as the return type of the current function, in case of `throw` the type must be derived from `System.Exception`.

Exceptions come in two forms, exceptions generated by the system and exceptions thrown by the user. Exceptions are caught by enclosing `try`, `catch` and `finally` blocks. Exception may be re-thrown to allow them to bubble up.

The order of events in jump statements can be: before and after `try` block, before and after jump statement with an optional before and after expression evaluation, and before and after `catch` or `finally` blocks. The `catch` event has an additional before and after code block. A system-generated exception does not have a before event.

The `return`, `throw` and `catch` events have one static property, namely:

Expression type Type of expression.

They have one associated dynamic property, which is available after the events given in Table 4.5.

Expression Value Value of expression.

Table 4.5: Jump events (one of `return`, `throw` or `catch`) and after which event the dynamic property is known

Property	Jump statement	Expression	Code block
Expression value	after	after	before

CHAPTER 5

Approach

This chapter discusses our approach for creating an event pattern language for Compose⋆. Before introducing the pattern language, we present several approaches to how events can be instrumented and monitored, and how events fit in the conceptual model of composition filters. Chapter 6 describes how the pattern language is integrated with the syntax of Compose⋆.

5.1 Event Instrumentation

Weaving is the process of composing the base system with aspects, as explained in Section 1.3.2 on Aspect Weaving. In an event-based join point model, pointcut expressions can be made that consist of one or more events. Each event should be signalled when it occurs. This is referred to as *event instrumentation*.

Like aspect weaving, the three most common approaches for event instrumentation are in the *source code*, in the *byte code*, or in the *run-time* environment. The same limitations as listed in Section 1.3.2 apply.

Weaving in an adapted run-time makes the base program oblivious to any aspects. Another advantage is the potential of dynamic deployment of aspects without recompilation. One of the goals for Compose⋆, however, is language independence. By modifying run-time environments, we have to keep up with new developments for each supported run-time.

Weaving hooks in either the source or the byte code removes the dependency on a adapted run-time at the cost of dynamic weaving. Also, dynamic events, like garbage collection and thread switching, might be unsupported if the run-time does not provide that information. Weaving in byte code, as opposed to source code, has the advantage of having support for all languages targeting that byte code.

The current weaver of Compose⋆ links concern code to the *base program* and weaves hooks for each possible invocation to this class, according to the weave specification. A weave specification is generated based on the concern

specification and information harvested from the base program assemblies. Each filter expression is evaluated at run-time to determine which filter action to take.

To remain both programming language and run-time independent, and to stay compatible with the current weaver, event hooks are also woven in the byte code. This means that for events, every pattern needs to be evaluated at each possible place in the code where that event might match. At those places, called *event join points*, instrumentation hooks are woven in. Hooks will signal event monitors to process the event and update the state of each event pattern. For handling of Compose⋆ specific events, like filter handling, the run-time module of Compose⋆ is instrumented.

5.2 Event Monitoring

For pattern recognition to take place, events need to be generated and monitored. Instrumented code will signal events by calling an *event monitor*. When the monitor receives an event, execution of the base program is suspended. The monitor determines at run-time if events match the conditions specified by the event patterns. Every pattern can be seen as a finite state machine where events signal state transitions. Upon reaching the final state, the monitor will invoke the associated advice, while the base program remains suspended. Advice can be implemented as a method inside an imposed class, or as internals and externals. The advice itself, just like the base code, can generate events for further matching of other patterns. After each pattern has been checked for possible transitions and each advice has executed, the base code is resumed again. Monitoring can take place on two levels: global monitoring and local monitoring.

Global Monitoring
By monitoring on a global level, each event is received by a global object which keeps state for all defined patterns. This can be program wide, or per thread. In the latter model, patterns which cross thread boundaries can not be checked.

Local Monitoring
The alternative is monitoring directly at the source of the event. Each object will have an associated event monitor. Events will only be matched locally. Local monitoring can be implemented in two ways. The state of a pattern can be remembered in between method calls or reset on each call. By remembering state, patterns can be defined to track the order of calls. This is useful for classes implementing a protocol. Take for example a class with methods for login, logout and write, where writing is only allowed for logged in users.

Related to local versus global monitoring is *encapsulation*. Encapsulation is about hiding internal representations and is often used with *data abstraction* to provide a strict external interface. An example is a private variable and controlling its access through getter and setter methods. The goal is to minimize code paths which can access and change data. Abstracting data behind a strict

interface allows changes to the internal representation without affecting externals. A global event monitor will receive all events of all objects and thus exposes the internal representation of those objects. As a consequence, a pattern will be tightly coupled to the events from those internal representations. A local monitor will only receive events from the local imposed object and match those to locally defined patterns. It does not expose its internals to other monitors. Upon refactoring a class, the only patterns affected are those patterns imposed on that class and depend on the changed internals.

Discussion

The join point model presented in this thesis is based on local monitoring, where state is remembered between calls to a monitored object. Restricting monitoring to local events, however, does not rule out patterns of object interactions. By having a local monitor emit a compound event on a successful match, other monitors will be able to track object interactions by matching on those composed events. This way, patterns abstract from internal representations of other objects.

For sending compound events, objects need to be arranged in a hierarchy. There are different strategies for determining such an arrangement. At compile-time by analyzing the abstract syntax tree (AST) to derive a class graph (interactions between classes). But when classes are instantiated multiple times, there is a problem of how to determine which instances to connect. At run-time there is a possibility to look at the stack and send the compound event up the call hierarchy. Event monitors associated to object instances remember state in between method calls. This allows for matches on a sequence of events, not only over the call chain. Another alternative is to create a registry and let the programmer have control over the hierarchy.

Sending compound events up an object hierarchy lies outside the scope of this thesis.

5.3 Event Filters

As described in Section 1.4.3, composition filters manipulate sent and received messages. Messages can be intercepted on method calls (by output filters) and method execution (by input filters). An ordered set of input filters and output filters make up a filter module. Filter modules are the units of reuse and instantiation of crosscutting behavior. Superimposition maps a filter module onto a set of object instances. The current framework selects objects at class level for superimposition. Figure 2.1 on page 15 depicts the components of the composition filters model. Every message received by an object is reified and passed through the incoming (or outgoing) filter sets. The message will continue to the next filter until discarded or dispatched. Dispatch will activate the, possibly modified, message and invoke it. Each filter can either accept or reject a message. The semantics of acceptance and rejection depend on the type of the filter as given in Section 2.2.

Incoming and outgoing messages, when instrumented, generate events. Events are processed by event monitors which try to match the event against one or more

patterns. There might be an analogy between message filters filtering messages, and event patterns filtering events. This suggests we can reuse the current framework of message handling and apply it to events. There are three alternatives for such an integration of event filters in the composition filters model. Either add event filters along with message filters to incoming filter sets, outgoing filter sets, or both. Each alternative is depicted in Figure 5.1. The solid arrows represent events or messages. When they originate from the interface part, they can be generated by inner methods or internals.

There are, however, drawbacks in grouping event filters with message filters. To address these, two more alternatives are discussed. They are depicted in Figure 5.2.

Event filter inside incoming or outgoing filter set

Figure 5.1a and 5.1b use the existing filter modules for processing both message events and other events. The first alternative mixes event filters in the incoming filter set, while the second alternative uses the outgoing filter set. This limits reasoning about message events to either incoming or outgoing messages and introduces ordering dependencies between event filters and message filters.

Event filter on both incoming and outgoing filter set

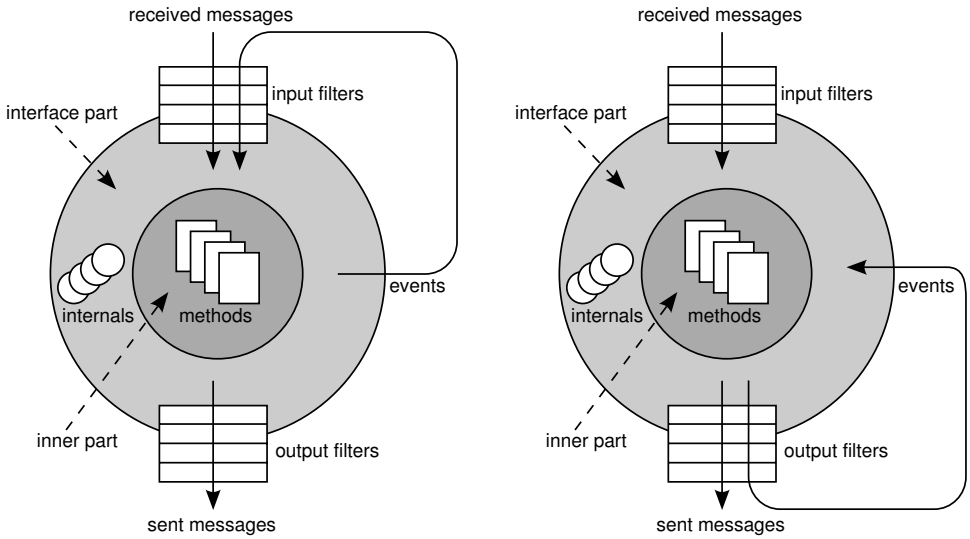
Figure 5.1c combines the previous two alternatives by using event filters in both incoming and outgoing filter sets. Generated events will go through both sets of filters. However, filters between both filter sets do not share state, thus cannot relate incoming messages to outgoing messages. Ordering problems between event filters and message filters still exist.

Event filter in separate filter set

Figure 5.2a uses a separate filter module for events. This makes it possible to share state between incoming and outgoing filters, and is thus able to relate both types of messages in a single event filter. Giving event modules priority above message filter modules will create discrepancy between the state of an event filter and the action taken by a modified reified message.

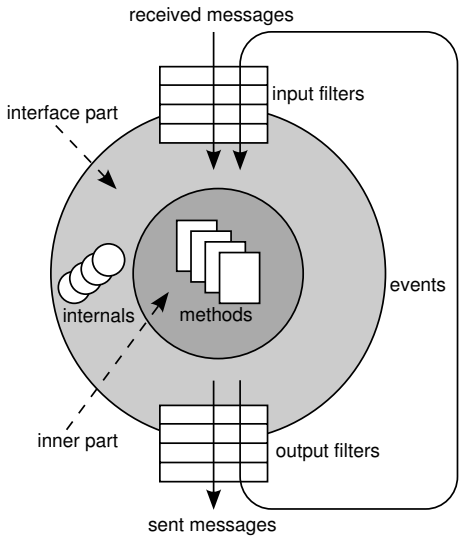
Separate events from messages

Another alternative is to allow an event filter to receive events from both the inner object and from the imposed filter modules. This enables us to define a pattern language which can handle any kind of event which is local to the imposed object. Figure 5.2b shows the resulting model. Events can be generated for filter handling, execution of a method on the imposed object, and execution of an internal referenced from a filter module. Examples of filter handling events are evaluation of a condition or modifying the reified message. In the figure they are represented by the events coming from the incoming and outgoing filter sets. This allows us to differentiate between the call and the possible executions of a method.



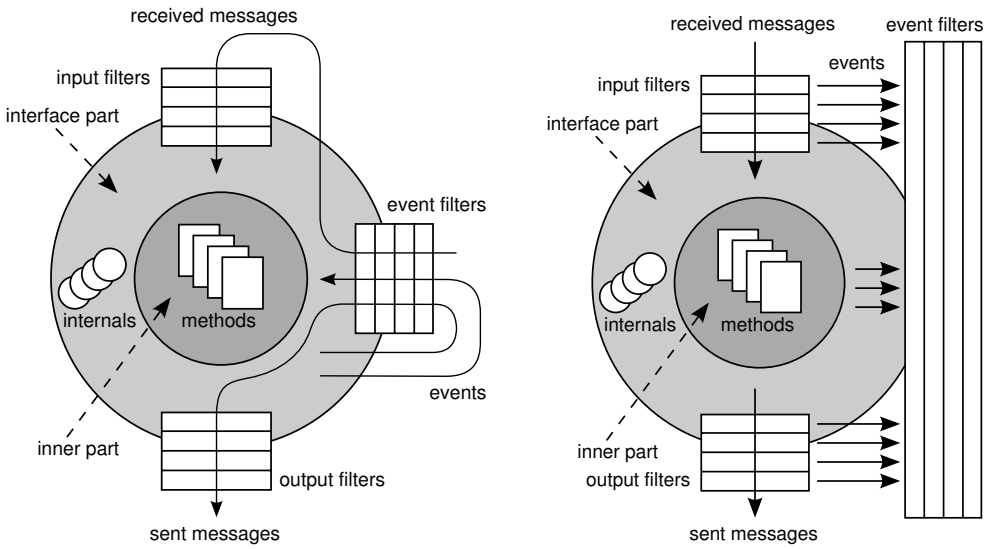
(a) Events go through input filters only

(b) Events go through output filters only



(c) Events go through both input and output filters

Figure 5.1: Alternatives for composition filters model extended with event processing, where event filters are mixed with message filters. For clarity, externals, condition methods, instance variables and superimposition have been omitted.



(a) Events and messages go through separate event filters first

(b) Events separated from messages

Figure 5.2: Alternatives for composition filters model extended with event processing, where event filters are separated from message filters For clarity, externals, condition methods, instance variables and superimposition have been omitted.

Discussion

Methods, in general, are made for a specific purpose, as formulated in a contract for example. Design by contract [Meyer, 1997] is a technique to help build better software by organizing the communication between software elements through specifying the mutual obligations and benefits that are involved in those communications. The specifications are called contracts. Pre-conditions are obligations that must hold before execution of a particular method, and post-conditions are benefits that are guaranteed on completion of that particular method.

Event monitors invoke advice when a certain behavior occurs. By grouping event filters with message filters, the relative order between filters matter. Dispatching a message before it reaches an event filter might have the consequence of keeping the event filter from ever matching its pattern. Even though, in general, the new method is assumed to uphold the same contract. The other way round is when a message is matched by a pattern, before being dispatched to another method. This can cause another pattern from matching its pattern. Suppose one pattern matches on a method invocation of instances of class `B`, and another pattern matches on instances of class `C`. When a method of class `B` is called, only the first pattern will match. But a message filter can alter the message and dispatch to class `C`.

Introducing a separate module with priority for event processing does not take this discrepancy away. The ordering of modules, however, can be made configurable by using ordering constraints as proposed by Nagy [2006]. In his

```
concern C {
  filtermodule fmA {
    // ...
  }
  filtermodule fmB {
    // ...
  }
  superimposition {
    selectors
      s = {C | isClassWithName(C, ...)};
    constraints
      s <- C.fmA before C.fmB;
  }
}
```

Listing 5.1: Example of filter module ordering in Compose⋆

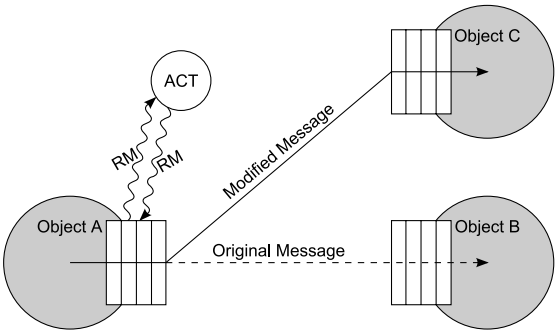
proposal, constraints are specified inside the superimposition part of a concern as demonstrated in Listing 5.1. Although these partial ordering constraints allow the developer to specify priorities, there is a conceptual difference between messages and events.

Objects communicate with each other by sending *messages*. A message is a request from one object instance (sender) to another object instance (receiver) to perform a method. A message contains parameters to provide information that is required for successful execution of the method. Message filters in Compose⋆ are able to intercept messages on both the sending and receiving side. An example flow of a message is depicted in Figure 5.3a, where a message is sent by object `A` to object `B`. The outgoing message, however, is filtered by a meta filter which modifies the reified message (RM) by changing the target to object `C`.

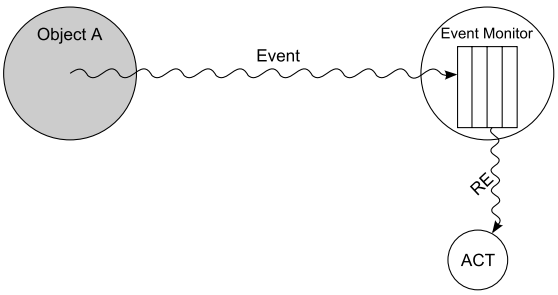
Events signal behavior of an object by transmitting a message to a local event monitor. On a successful match, advice (implemented as ACT) is invoked with the last reified event (RE) as parameter. This is depicted in Figure 5.3b. Events are used for monitoring the program and, unlike messages, cannot be modified. They are an atomic message of what is about to happen (*before events*), or what just happened (*after events*).

The question remains where event processing should take place when filters alter a message. By extending the join point model, filter processing can be instrumented for events. This allows a pattern to specify whether certain method invocations should be matched before or after filter modules. This corresponds to the model depicted in Figure 5.2b, where event modules are put alongside an object and all its imposed filter modules. This is the model we picked for integration of events in Compose⋆. If in the future it is needed to exclude certain filter modules for event filters, the imposition mechanism can be changed to only impose on the base object and explicitly named filter modules.

In practice, though, it is best to keep patterns independent of a particular execution path. An event pattern should be described in terms of events which contribute to its goal and do not depend a particular method name. This creates loosely coupled event patterns which can be reused easily.



(a) Message flow example



(b) Event flow example

Figure 5.3: Flow of messages compared to events

5.4 Event Pattern Language

To support crosscutting beyond the level of object interfaces, we create a fine-grained join point model (JPM) as described in Section 4.3, and provide a pointcut language to relate event join points to one another. This pointcut language should be expressive enough to solve common problems. The example presented in Section 3.1 discussed two problems, the arranged patterns problem and jumping aspects. We use the same example to illustrate our language and show a solution for the two problems.

5.4.1 Example Pattern

The primary concern of class `Inventory`, as presented in the example of Section 3.1, is keeping track of an inventory of products. The inventory is stored inside a private variable `stock` of type `ArrayList`. Whenever this variable changes, the state changes, and all registered observers should be notified.

Suppose our notify concern is specified as follows. When adding or removing a single product, notify all observers. Also, when adding or removing a list of products, notify only after the operation as a whole is finished.

Each state changing method results in a call to one or more state changing methods of variable `stock` of class `Inventory`. Such calls can be either direct or indirect calls. The bulk updater of Listing 3.7 is an example of an indirect call to `Add` through method `AddProduct`. For simplicity we only consider methods `Add`

Table 5.1: Regular expressions and their corresponding regular sets

Expression	**Set**	**Meaning**
a	$\{a\}$	Instance
(a)	$\{a\}$	Grouping
$a \mid b$	$\{a\} \cup \{b\} = \{a, b\}$	Alternative composition
ab	$\{a\}\{b\} = \{ab\}$	Sequential composition
$a?$	$\{\epsilon\} \cup \{a\}$	Optional instance
a^*	$\{a\}^*$	Kleene star, zero or more instances
a^+	$\{a\}\{a\}^*$	Kleene plus, one or more instances
a^n	$\{a\}^n$	n sequential compositions, n is fixed

and `Remove` of class `ArrayList` as state changing methods. One can imagine a library of such annotations is made available for system classes.

To help formulate our pattern, we identify the relevant events and introduce a symbol as shorthand for reference. Symbols e_f and r_f will represent events for execution and return of a method of class `Inventory`. And symbols c_l and r_l will represent events for a call and return to a state changing method of class `ArrayList`. Using these symbols, we formulate a regular expression. Most people are familiar with regular expressions, search tools like 'grep' and languages like 'perl' make extensive use of them. The following regular expression captures all methods which call a state changing function before returning to the caller.

$$e_f(c_l r_l)^+ r_f \tag{5.1}$$

The operator + indicates that the pattern in parenthesis can be repeated one or more times.

A regular expression will be matched at run-time by an event monitor. All events received by an event monitor can be seen as the input string against which the regular expression is matched. Upon a successful match, the notify advice should be executed.

5.4.2 Regular Languages and Traces

A language is defined as a set of strings over an alphabet Σ. These sets are regular if they can be built from the empty set $\emptyset$, the set containing the null string $\{\epsilon\}$, and the sets containing a single element of the alphabet ($\{\alpha\}$ for every $\alpha \in \Sigma$) using the operations of union $X \cup Y$, concatenation XY, and Kleene closure X^*, where X and Y are regular sets over Σ. Regular expressions are used to abbreviate the descriptions of regular sets [Sudkamp, 1997]. Regular expressions can make use of the operations listed in Table 5.1.

The events defined in Section 4.3, all belong to our alphabet Σ. When the base program runs, all signalled events form a sequence, or *trace*. The set of all finite sequences of events of Σ are denoted by Σ^*. Thus, a trace is a subset of Σ^*.

When matching a pattern, we are often only interested in a trace restricted to events of interest and *skip* all other events. Suppose we are interested in events A and define a pattern over those events. The pattern will be matched against

the restricted trace $tr \upharpoonright A$.

$$\begin{aligned} \Sigma &= \{a, b, c, d\} \\ A &= \{b, c\} \\ tr &= \langle b, c, b, b, a, c, d, a \rangle \\ tr \upharpoonright A &= \langle b, c, b, b, c \rangle \end{aligned}$$

5.4.3 Matching Process

The matching process of regular expressions can be visualized by automata, where circles represent states and arrows represent transitions between states. The matching process starts when an event is signalled by the instrumented code. An event is received by a local monitor which will process the event against each pattern imposed on the local object.

When no previous pattern matching process has started, a new pattern is instantiated. Between events, each pattern instance remembers its state. An event is checked for a transition against all active pattern instances. Each event, however, could be the start of a new pattern sequence, thus, each event is also checked against a new pattern instance. When a pattern reaches an accepting state through a transition triggered by a declared symbol, event filters are evaluated for advice execution. When a pattern reaches a state with no outgoing transitions, the pattern terminates.

Regular expression 5.1 corresponds to the automaton of Figure 5.4. To make state transitions within the automaton total[1], *skip* and $fail$ labels are added. Transitions marked with label *skip* represent all events in which we are not interested. Transitions marked with label $fail$ represent those events which are not allowed to happen in a successful match, but are in the alphabet of symbols. Label $fail_1$, for example, represents the symbols c_l, r_l and r_f, as symbol e_f is the only one allowed. If an accepting state (5) is reached, but not through *skip*, the notify advice is executed. If the fail state (0) is reached, our pattern failed to match the input string and terminates. The semantics of skip and fail, however, have a big impact on the matching process. Two alternative semantics for skip and fail are discussed next.

[1]Transitions can be seen as a *total* function, thus defined over all states and all symbols.

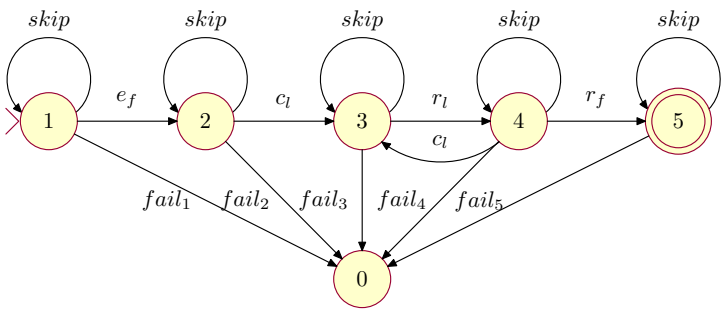
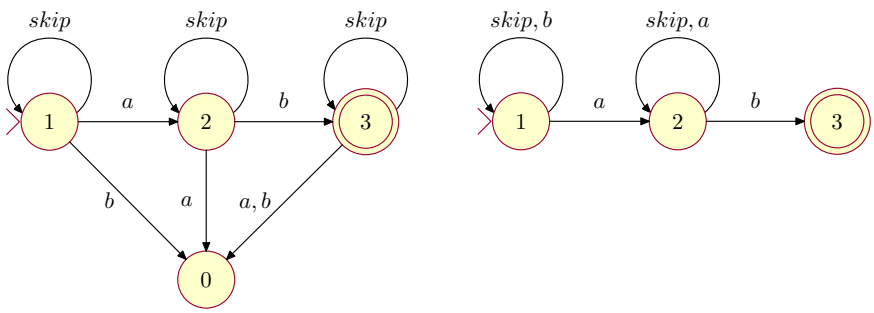


Figure 5.4: Finite automaton for notifications



(a) Implicit fail model (b) Explicit fail model

Figure 5.5: Pattern ab over set $\{a, b\}$

In the first alternative, label *skip* contains any event not declared as a symbol. It is represented by a self-loop in an automaton. Transitions to the $fail$ state (0) are added for each declared symbol not present in the pattern at that particular state. We will refer to this alternative as the *implicit fail model*. The second alternative adds those symbols to label *skip* for a particular state. Transitions to the $fail$ state only contain those symbols explicitly excluded using a negation operator ($\neg$). We will refer to this alternative as the *explicit fail model*. The explicit fail model corresponds in syntax to extended regular expressions (ERE). Automata created for the same pattern in both models are not equivalent, however. Combining them with the pattern instantiation mechanism described earlier, both models become an alternative for matching events on traces. Both alternatives, however, have their advantages and disadvantages. They will be explained by putting both models side-by-side using simple examples.

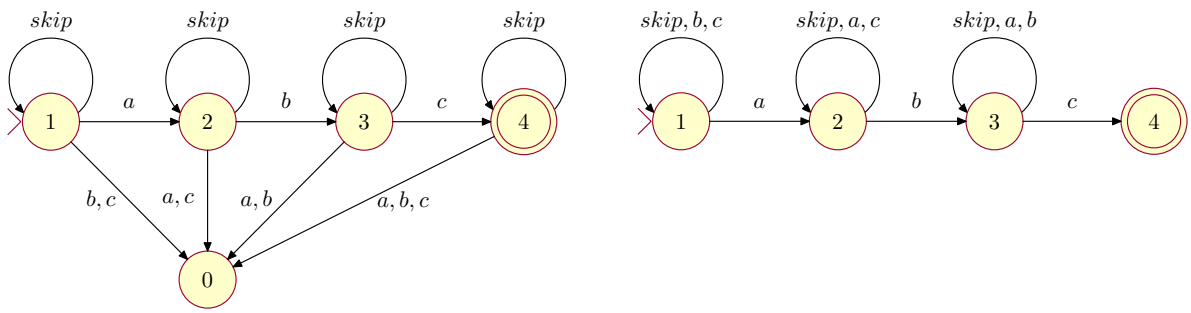
Example One

Suppose we have an object with two methods, `a` and `b`, and want to match for pattern ab. We restrict our alphabet to $\{a, b\}$. Figure 5.5 depicts the resulting automaton for both models. Suppose at run-time our trace is $\langle a, a, a, b, b, b\rangle$.

The automaton of Figure 5.5a will match one time. When the first a is processed, the automaton will be in state 2. The second a will lead to the fail state 0 and the automaton terminates. But an event is also checked for a match from the initial state, the new pattern instance will also be in state 2. The same happens on the third a. On the first b, we reach an accepting state and terminate. The second and third b go directly from the initial state to the fail state.

The automaton of Figure 5.5b, on the other hand, will match three times. When the first a is processed, the automaton will be in state 2. When the second a is processed, the automaton will follow the self-loop. But the event might be the start of a new pattern, thus the second pattern instance is also in state 2. The same happens on the third a. The first, second and third b will terminate the patterns by reaching the accepting state.

The implicit fail model will only accept a match on the third a and first b, while the explicit fail model will keep a pattern on hold by ignoring all irrelevant events.



(a) Implicit fail model (b) Explicit fail model

Figure 5.6: Pattern abc over set $\{a, b, c\}$

Example Two

We extend the object from the previous example with an extra method `c`. The pattern to match is abc with alphabet $\{a, b, c\}$. This time our trace is $\langle a, c, b, c\rangle$. Figure 5.6 depicts the resulting automaton for both models.

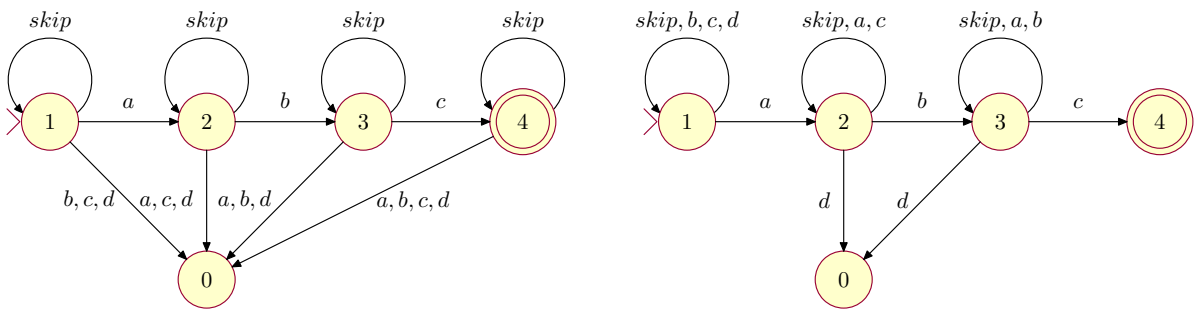
The automaton of Figure 5.6a will not have any match. The second event c, and all subsequent events from the trace, will cause the automaton to fail. The automaton of Figure 5.6b, on the other hand, will match once. It will ignore the second c event by following the self-loop. All subsequent events will take the automaton to the accepting state.

To get the second alternative to fail, we have explicitly disallow a c event after an a event. The resulting pattern is $a\neg cbc$. It will add fail state 0 to Figure 5.6b with a transition from state 2 on c. The implicit fail model can not define a pattern which both includes and excludes the same type of event, unless we introduce a negation operator at the cost of less clear semantics.

Example Three

We extend the object from the previous example with an extra method `d`. The pattern to match is again abc, but this time without a d. The alphabet for the pattern is $\{a, b, c, d\}$

In the implicit fail model, it is enough to define the pattern as abc. Any event declared as symbol, but not referred to in the pattern, is considered to be an implicit fail. In the explicit fail model, we have to exclude all d events after each other event in the pattern. The resulting pattern is $a\neg db\neg dc$. Figure 5.7 depicts the resulting automaton for both models. While both automata ignore d events, they are not equivalent with respect to handling multiple occurrences of events which are allowed. Consider the following trace $\langle a, b, a, c\rangle$. The automaton of Figure 5.7a will fail to match, while the automaton of Figure 5.7b matches once.



(a) Implicit fail model (b) Explicit fail model

Figure 5.7: Pattern abc with no d over set $\{a, b, c, d\}$

Discussion

The semantics for skip and fail states are an important language design decision. They influence the matching process of patterns and thus have to be kept in mind when constructing patterns. The decision of choosing one model over the other can only be made when more domain problems are analyzed. In the current context it is unclear which model will describe intended behavior best, without knowing what the symbols stand for and which problem one is trying to solve.

Explicit exclusion clutters a pattern with events in which one is not interested. The pattern becomes quite lengthy, especially when more than one event is excluded. The implicit fail model describes a pattern in terms of what is allowed to happen, but might cause unintended failures when a developer does not notice an unused symbol in his pattern. This can, however, be circumvented by tool support. An informative warning can be generated at compile-time.

To keep the number of operators in the language low, we decided on using the *implicit fail model*. The inventory example does not require explicit exclusion of events. If the need arises for more expressive power, it can be added later on.

5.4.4 Extending Regular Expressions with Counters

The automata used throughout this chapter are examples of deterministic finite automata (DFA). A DFA is a read-once machine in which the instruction to be executed is determined by the state of the machine and the input symbol being processed. Kleene's theorem shows that finite automata accept precisely the languages generated by regular grammars [Sudkamp, 1997, page 203].

But not all problems are solvable with a regular language. The event-based join point model introduced in Section 4.3 contains separate events for method call and method return. If we want to match on the last return from a recursive call, we need to balance the number of calls and returns.

$$\mathcal{L} = \{c^i r^i \mid i > 0\} \tag{5.2}$$

Such a pattern can not be described by a regular language, unless the iteration level is fixed. This is shown by using the *pigeonhole principle*. Assume a machine

$\mathcal{M}$ exists with k states to check that the number of calls and returns are balanced. Any path of length k or more contains a cycle. Now consider the input string z consisting of k calls and k returns, which means that at least one state must be visited multiple times. The cycle allows for a string to be pumped multiple times. The *pumping lemma* is based on this principle to prove a language is not regular. We break input z up in three parts uvz, where v represents the cycle in the number of calls. The pumping lemma states that if $\mathcal{L}$ is regular the following must hold [Sudkamp, 1997, pages 212–216]

i) length$(uv) \leq k$;
ii) length$(v) > 0$;
iii) $uv^p w \in \mathcal{L}$ for all $p \geq 0$.

To satisfy these conditions $u = c^i$, $v = c^j$, and $w = c^{k-i-j}r^k$, where $i + j \leq k$ and $j > 0$. By pumping v we get $uv^2w = c^i c^j c^j c^{k-i-j} r^k = c^k c^j r^k$, which is not in $\mathcal{L}$. A contradiction, thus $\mathcal{L}$ is not regular. To accept this language, the machine needs the ability to record the number of calls made. Push down automaton (PDA) extend a finite automata with a stack. Stack operations affect only the top item of the stack; a push places an element on the stack and a pop removes the top element.

An example of a pattern using a counter is the last call to a recursive implementation of quicksort. The pattern will be the same as expression 5.2. Using a counter is not restricted to recursive calls. Suppose we have a method that needs to perform two tasks (a and b) on every item of an array. Where task b can only start once task a is finished. We call the method e_f and its return r_f.

$$\mathcal{L}_1 = \{e_f a^i b^i r_f \mid i \geq 0\} \tag{5.3}$$

Suppose we add a third task c, to be performed when b finishes. We modify the expression accordingly.

$$\mathcal{L}_2 = \{e_f a^i b^i c^i r_f \mid i \geq 0\} \tag{5.4}$$

This expression, however, is *not* context-free anymore. This can be shown by applying the pumping lemma for context-free languages [Sudkamp, 1997, page 244]. Intuitively one can verify this by pushing an item on the stack for every a and popping it for every b. This ensures that the number of executions of a and b are equal. But for c the stack is empty, thus provides no means to ensure that it is performed an equal number of times. We will implement counters by associating a separate counter for every occurrence and verify they are the same before accepting a run.

5.5 Temporal Logic

Using a notation based on regular expressions has the benefit of having a low learning curve. Most programmers are familiar in using them. There are, however, other formalisms for expressing behavior. We briefly discuss temporal logic.

Temporal logic is typically used in model checking. Model checking is an automatic verification technique for finite state systems. Specifications are written in propositional temporal logic and verification is done by an exhaustive search of all states in the model.

There are two main kinds of temporal logic: linear time and branching time. They differ in how time is modelled. In linear time every state has one successor state, in branching time every state can have more than one possible successor state. For our purpose we only want to verify a single program execution at a time, and not quantify over all possible paths, which is why we restrict the discussion to linear time.

Linear temporal logic (LTL) is a modal linear time logic over infinite traces. It was introduced by Pnueli [1977]. The syntax of LTL is as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \psi \mid \bigcirc\,\phi \mid \phi\,\mathcal{U}\,\psi$$

where p is an atomic proposition. The meaning of the formulae are as follows: negation ($\neg\phi$ holds if ϕ does not), disjunction ($\phi \vee \psi$ holds if either ϕ holds or ψ holds), next ($\bigcirc\,\phi$ holds in a state, if ϕ holds in the successive state), and until ($\phi\,\mathcal{U}\,\psi$ holds if ψ eventually will hold for some state and ϕ continuously holds until that state). The following operators can be derived:

$$\begin{aligned}
true &\equiv p \vee \neg p \\
false &\equiv \neg true \\
\phi \wedge \psi &\equiv \neg(\neg\phi \vee \neg\psi) \\
\phi \Rightarrow \psi &\equiv \neg\phi \vee \psi \\
\phi \Leftrightarrow \psi &\equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
\Diamond\,\phi &\equiv true\,\mathcal{U}\,\phi \\
\Box\,\phi &\equiv \neg\,\Diamond(\neg\phi) \\
\mathcal{W}\phi &\equiv \Box\,\phi \vee (\phi\,\mathcal{U}\,\psi)
\end{aligned}$$

Where the first five definitions (true, false, conjunction, implication, and equivalence) are standard propositional logic. The last three operators are future (eventually ϕ), globally (always ϕ), and unless (weak until ϕ). Formula $\Diamond\,\phi$ holds if either ϕ is true now, or it will become true in some state in the future. Formula $\Box\,\phi$ holds if ϕ holds in the current state and it holds in every state in the future. Formula $\mathcal{W}\phi$ holds if $\phi\,\mathcal{U}\,\psi$ holds, without guarantee that ψ will ever hold, meaning that ϕ continuously holds if ψ never holds.

Temporal properties are mainly classified into *safety* properties and *liveness* properties. A safety property asserts that "something bad" never happens in future ($\Box(\neg bad)$), while a liveness property asserts that "something good" eventually happens in future ($\Diamond(good)$). In model checking, temporal properties are

translated to finite Büchi automata over infinite words. The system to be checked is also described by such an automaton. Let $\mathcal{B}_{spec}$ be the Büchi automaton for the specification of the system and $\mathcal{B}_{sys}$ be the Büchi automaton modelling the system. Then language $\mathcal{L}(\mathcal{B}_{spec})$ is the set of allowed behaviors of the system. The system satisfies the specification if

$$\mathcal{L}(\mathcal{B}_{sys}) \subseteq \mathcal{L}(\mathcal{B}_{spec}) \tag{5.5}$$

The inclusion specifies that the behavior of the system is allowed by the specification. By taking the complement of the language of the specification, we can formulate 5.5 as

$$\mathcal{L}(\mathcal{B}_{sys}) \cap \mathcal{L}(\overline{\mathcal{B}_{spec}}) = \emptyset \tag{5.6}$$

If the intersection is not empty, each infinite word represents a counter example.

In run-time monitoring, a particular execution trace of the program is verified against a specification, instead of searching the whole state space. This implies a finite trace. One has to define how operators like global and future operate on finite traces. A finite trace semantics for LTL is described by Giannakopoulou and Havelund [2001] and by Havelund and Roșu [2004].

Temporal logic is often used to specify invariants and pre- and post-conditions for design by contract. Invariants specify properties which should always hold, pre-conditions specify properties which have to be satisfied before a certain state, and post-conditions specify what should be done after a certain state. Pre-conditions are usually specified in past temporal logic, which is just as expressive as future temporal logic.

It is, however, difficult to write an LTL expression corresponding to regular expression 5.1 to notify on changes. The recursion can only be specified as a countable number of steps, because LTL can only specify star-free ω-regular languages [Kupferman et al., 2001]. The nesting level of the expression will be deep.

The notify concern would look like $\Box(change \Rightarrow \bigcirc notify)$. Where *change* should be an expression for entering an execution (e_f) with calls to inventory changing functions (c_l). An expression as $\Box(e_f \Rightarrow \Diamond r_f)$ will imply that an execution is eventually followed by a return. This formula, however, is also satisfied by r_f without e_f. Creating a similar formula for c_l and r_l and combining the two formulae needs explicit ordering between r_l and r_f. One can specify a trace of one inventory function with $\Box((e_f \wedge \bigcirc(c_l \wedge \bigcirc(r_l \wedge \bigcirc r_f))) \Rightarrow \bigcirc notify)$. The nesting and explicit next and conjunctions operators make the expression less readable than a language based on regular expressions. There are, however, various temporal logics which add regular operators to temporal logic, like extended temporal logic (ETL) [Kupferman et al., 2001].

CHAPTER 6

Integration

To specify event patterns inside concern specifications, the pattern language introduced in Section 5.4 is integrated with the current syntax of Compose⋆. This chapter explains how concerns are extended with event modules. To illustrate event modules, a solution for the inventory example is presented. Followed by a description how event modules are made reusable, and how conflicts between event modules can be resolved.

6.1 Event Modules

A concern, as described in Section 2.2, can be divided in three parts: filter modules, superimposition and implementation. To support events, a new *event modules* part is added.

Event modules, like filter modules, are the units of reuse. They can contain six parts: internals, externals, conditions, symbols, a pattern, and event filters. An example of an event module is given in Listing 6.2 in the next section. The superimposition part is extended to support local monitoring of event modules. It binds an event module to objects, including all filter modules imposed on those objects. An example of superimposition is also shown in Listing 6.2. Each part of an event module is discussed next. The formal grammar is given in Section 7.1.

Internals and Externals

Internals and externals in event modules serve the same purpose as in filter modules. To declare objects whose methods may be invoked as advice, as defined by the event filters section. Examples are:

```
internals
  notifier: ExampleAOP.Notify;
externals
  logger: ExampleAOP.Logger = ExampleAOP.Logger.Instance();
```

Internals are instantiated per event module instance. Externals use an initialization expression which should return an object instance, it is usually implemented as singleton. Multiple internals and externals may be declared.

Conditions

Conditions are methods that offer an abstraction of the state of the implementation object. They must be implemented as side-effect free boolean expressions. Conditions are referenced by event filters to conditionally accept advice execution after a pattern matches. The following condition declaration refers to method `Observed` of internal `notify`.

```
conditions
  observed: notify.Observed();
```

Multiple conditions may be declared.

Symbols

The symbols section declares the restricted set of events used in pattern matching. It consists of a symbol identifier followed by an event modifier (`before` or `after`) and an event name. Multiple symbols may be declared, each ending with a semicolon. An event may be restricted to certain conditions on its properties. Events and their properties are described in Section 4.3. To relate two or more events together, variables are introduced.

Variables can be either free or fixed, depending on how they are declared. If declared as (event) module parameter, its value is fixed during the lifetime of the module. If declared as property in the symbols section, it is considered free. Variables are declared with a question mark and an identifier (`?x`). Whenever the associated event occurs, according to the pattern, and the variable is used on the left hand side of an assignment operator (`:=`), the variable is bound to the value of the right hand side. Variables (both free and fixed) are checked for equality and inequality by using the comparison operators (`=` and `!=`).

There are two types of module parameters, lists and single values. If declared (and referenced) with one question mark and an identifier, they represent a single value (`?y`). If declared (and referenced) with two question marks and an identifier, they represent a list (`??ys`). One can restrict an event property or a value to an item from a list by using a colon, `?y:??ys` for example. Identifiers for a variable are unique within a module, thus `?x:??x` is not valid.

The following symbols section declares four symbols:

```
symbols
  Ef: before execution(?f:=selector, sender!=self);
  Rf: after execution(selector=?f, sender!=self);
  Cl: before call(?l:=selector:??ls, target=inner.stock);
  Rl: after call(selector=?l, target=inner.stock);
```

Symbol `Ef` checks for any method execution that is called from outside the object, and binds the name (selector) to free variable `?f`. Symbol `Rf` checks for the associated return event by comparing variable `?f` with the selector of the event. Symbol `Cl` checks for any call to a state changing function on object `inner.stock` . The selector is bound to variable `?l` and can be any method name which is

Table 6.1: Pattern expressions and their notation inside event modules

Expression	**Notation**	**Meaning**
a	`a`	Instance
(a)	`(a)`	Grouping
$a \mid b$	`a|b`	Alternative composition
ab	`a b`	Sequential composition
$a?$	`a?`	Optional instance
a^*	`a*`	Kleene star, zero or more instances
a^+	`a+`	Kleene plus, one or more instances
a^n	`a[n]`	n sequential compositions, n is fixed
a^i	`a[?i]`	i sequential compositions, i is variable

part of set `??ls`. Symbol `Rl` checks for the associated return event by comparing variable `?l` with the selector of the event.

Pattern

A pattern consists of a pattern identifier and an expression over declared symbols. An expression is enclosed in curly brackets and ending with a semicolon. Any declared symbol not used in the pattern is not allowed to occur in a successful match. Because of this, an event module can only contain *one* pattern. To have multiple patterns, one has to define separate event modules. It is allowed to superimpose them on the same classes.

The notation used in all previous pattern expressions is not suitable as syntax inside concern specifications. The parser of Compose★ expects plain text files as input. Table 6.1 shows the equivalent notation in plain text for each operator. The following example demonstrates two patterns in that notation.

```
change: {Ef (Cl Rl)+ Rf};
fifth: {Ef (Cl Rl)[5]};
```

The first pattern specifies one or more occurrences of `Cl Rl` between `Ef` and `Rf`, while the second pattern matches on exactly five occurrences after one `Ef`.

When adding counters to a pattern expression, all symbols annotated with the same counter should occur an equal number of times. The counter uses the same notation as free variables and should have a unique identifier. For acceptance of a pattern with a counter, an expression should be given which should be satisfied by the counter. The expression is separated from the pattern with a vertical bar. The only expressions supported are zero or more occurrences (`>=0`) and one or more occurrences (`>0`). When the need for more complicated counter handling arises, other operators can be added.

```
recurse: {Ef Cl[?i] Rl[?i] Rf | ?i >= 0};
```

This pattern matches any number or recursive calls to `Cl`, including no call, from within method execution `Ef`.

Event filters

If a pattern reaches an accepting state, some advice is invoked as defined by event filters. Event filters can contain four parts: an identifier, a filter type, a condition and a target. The target can be any method defined in an internal, external or server (the object enhanced with filters). The condition part is optional and used to accept or reject advice. The default condition evaluates to true. The filter type can be any predefined filter. However, monitoring is a read-only operation, a filter can not alter an event. From the current set of predefined filters, dispatch, send and substitute can not be used as event filter. Substitute is a filter to explicitly modify properties (of a message). Dispatch and send invoke a reified message, but an event signals that something happened or is about to happen, thus can not be invoked. Other predefined filters, like meta, error, or any custom filter can be used. Filters have access to the last event that caused the pattern to reach an accepting state. An extension could be made to store the whole history, but one of the benefits of using state machines is that the history is embedded in the state.

Meta If the pattern reaches an accepting state and the condition evaluates to true, the (reified) event is sent as parameter to the target; otherwise the next filter is evaluated. The target that receives the reified event can base its actions on it. The reified event is read-only, contains all properties for that event, and does not have methods like proceed or reply. After the object finishes its execution, the next filter is evaluated.

Invoke The semantics of this filter are the same as for the meta filter, except that the invoked method does not receive a reified event as argument.

Error For messages, an error filter raises an exception if the filter rejects the message; otherwise the next filter is evaluated. For events, the filter set is evaluated when the pattern reaches an accepting state. To keep the semantics of the error filter the same, the condition is the only part left which can reject. If the pattern reaches an accepting state and the condition evaluates to false, an exception is raised; otherwise the next filter is evaluated. Useful in describing patterns of behavior that is not allowed to happen.

Multiple event filters may be defined by using the *filter composition operator* (`;`). They are evaluated in the order of declaration. Please note, the last filter does *not* end with a filter composition operator.

An example event filters section is:

```
eventfilters
  logger: Meta = {c1 => logger.Log};
  abuse: Error = {c2};
  notify: Meta = {c3 => notifier.Notify}
```

When the pattern reaches an accepting state, the event filters are evaluated one by one in the specified order. If condition `c1` evaluates true, method `Log` is called. If condition `c2` evaluates false, an exception is raised. If condition `c3` evaluates true, method `Notify` is called.

Discussion

This section described one way of integrating a pattern language with the syntax of Compose⋆. Another possibility would be to specify a pattern as an ordered set of filters. Analogue to how message filters match messages, event filters would match events. To be able to specify certain properties on events, the language for specifying the matching part of a filter must be extended.

A proposal by Doornenbal [2006] extends the filter syntax with properties. For messages, an example of a filter is `Dispatch = {condition, target=Inventory, selector:=Attach}`. The condition part, matching part and substitution part are collapsed into a single part, separated by commas. The evaluation order is from left to right. Properties are matched for equality with operator `=`, and assigned a value with operator `:=`. Adding an event type in front of properties, allows specifying properties on arbitrary events. Properties on events are read-only, though. By adding extra filter operators, for example `+` for Kleene plus, a pattern can be specified inside a filter. Properties on the same event (or message) have to be grouped, which can be done by introducing a grouping operators `<` and `>`. For example: `Meta = {condition, <Message.selector=AddProduct>, <Message.selector=Add,Message.target=stock>+}`. Besides the pattern, an event filter must specify which advice to invoke on a successful match. We can either reintroduce a second part to the filter, or, add a message (using an assignment operator) behind the pattern. An example pattern would be `Meta = {condition, <..., ...>, <Message.target:=notifier, Message.selector:=Notify>}`, which evaluates a condition, some event properties, and sends a message to `notifier.Notify`. However, the downside of specifying a pattern inside a filter, is that filters become quite verbose, which does not look good from a readability point of view.

Further more, pattern expressions which refer to the same event twice (i.e., *aba*), require both events to be fully specified. Introducing a separate section to declare event symbols removes this redundancy. This changes a pattern definition to describing behavior over an alphabet of symbols, where each symbol is declared in terms of properties over an event type. Which brings us to the language proposal made in this section.

The implicit *not* operator, discussed in the implicit fail model of Section 5.4.3,

```
eventmodule InventoryUpdate {
  symbols
    Ef: [change,fifth] before execution(?f:=selector, sender!=self);
    Rf: [change] after execution(selector=?f, sender!=self);
    Cl: [change,fifth] before call(?l:=selector:??ls, target=inner.stock);
    Rl: [change,fifth] after call(selector=?l, target=inner.stock);
  patterns
    change: {Ef (Cl Rl)+ Rf};
    fifth: {Ef (Cl Rl)[5]};
  eventfilters
    notify: Meta = {[change] notifier.Notify}
    logger: Meta = {[fifth] logger.Log}
}
```

Listing 6.1: A proposal for specifying two patterns in one event module

limits the number of patterns in an event module to one. To allow for multiple patterns inside one event module, extra syntax is needed to associate symbols and event filters with a pattern. Listing 6.1 shows an example by adding a pattern name in square brackets. It is unclear though, what the intended effect should be on free variables. Should the value be shared by both patterns, or should each pattern have its own binding? Sharing the value introduces side-effects when both patterns set the value. Limiting the number of patterns to one, avoids the issue, keeps syntax simple and allows for independent reuse of event modules.

6.2 Inventory Example using Events

This section presents a solution to our example problem, as discussed in Chapter 3, about notifications on state changes in instances of class `Inventory`. The solution is shown in Listing 6.2 and uses the syntax presented in the previous section.

Concern `InventoryUpdates` defines an event module `InventoryChange` (line 2). The event module defines a pattern, called `change` (line 9), consisting of four symbols (lines 4–7). The pattern attempts to match on the execution of a method of class `Inventory`, which calls state changing functions on variable `stock` before returning. The methods that are called from within the local object are ignored. The pattern corresponds to the example pattern described in Section 5.4.1. The pattern expression could actually be shortened to `Ef Cl+ Rf` with symbol `Cl` declared as `before call(selector=??l, target=inner.stock)`. This will match any call in set `??l` irrespective of recursive and non-recursive calls. Variable `stock` is an instance of class `ArrayList`. Using filter module parameters, the definitions of the state changing functions are decoupled from the pattern. Upon a successful match a message is sent to method `Notify`, which is defined in another filter module. By setting the target to `server`, we invoke the message on the 'enhanced' object, i.e., the concern with filters.

This pattern solves the *arranged pattern* problem and the problem of *jumping aspects* that were discussed in Chapter 3. The arranged patterns problem is prevented by abstracting away from the name of the method. The problem of jumping aspects is solved by checking the sender property of the method execution to select on those methods called from outside the local object. This captures the single state changing method `AddProduct` and the bulk updated `AddProducts`.

Listing 6.3 shows the concern `InventorySubject` which defines the interface to attach and detach observers, and to notify them. Methods `Attach` and `Detach` are called by an observer. Method `Notify` is invoked by the event module described above.

Listing 6.4 shows the superimposition specification. Both the event module and the filter module are superimposed in every instance of class `Inventory`. The selector language, explained in the M. Sc. thesis of Havinga [2005], is also used to select the state changing methods of class `System.ArrayList` (lines 5–8). It is based on the predicate language Prolog. Had class `ArrayList` been annotated, we could have used the predicate `methodHasAnnotation` instead of `isMethodWithName`.

```
concern InventoryUpdates {
  eventmodule InventoryChange(??ls) {
    symbols
      Ef: before execution(?f:=selector, sender!=self);
      Rf: after execution(selector=?f, sender!=self);
      Cl: before call(?l:=selector:??ls, target=inner.stock);
      Rl: after call(selector=?l, target=inner.stock);
    pattern
      change: {Ef (Cl Rl)+ Rf};
    eventfilters
      notify: Invoke = {server.Notify}
  }
}
```

Listing 6.2: Event module to detect state changes on variable `stock`

```
concern InventorySubject {
  filtermodule SubjectInventoryFilter {
    internals
      subject: Subject;
    inputfilters
      attach: Dispatch = {[*.Attach] subject.Attach};
      detach: Dispatch = {[*.Detach] subject.Detach};
      notify: Dispatch = {[*.Notify] subject.Notify}
  }
}
```

Listing 6.3: Filter module which defines an interface for observer registration and notification

```
concern WeaveInventoryObserver {
  superimposition {
    selectors
      subjects = {C | isClassWithName(C, 'Inventory')};
      arraychange = {F | isClassWithName(C, 'System.ArrayList'),
                         classHasMethod(C, F),
                         (isMethodWithName(F, 'Add');
                          isMethodWithName(F, 'Remove'))};
    eventmodules
      subjects <- SubjectInventoryFilter,
                  InventoryChange(arraychange);
  }
}
```

Listing 6.4: Superimposition specification binding an event pattern and a filter module to instances of class `stock`

By using filter module parameters and keeping event patterns generic, superimposition becomes the 'glue', composing all parts together. This introduces a notion of sharing event modules, the next section explores this further.

6.3 Abstract Event Modules

Sharing patterns between event modules, requires sharing of symbols. Sharing symbols between unrelated patterns is questionable, because symbols are closely bound to a pattern, especially if they contain free variables and an implicit not (when a pattern does not reference a symbol).

If a developer wants to reuse event modules, there are three options available: superimposition, pattern reuse, and event filter reuse. If the module is generic enough, an event module can be superimposed on other objects. Reusing a pattern with different advice is achieved by referring to a pattern in another event module and defining new event filters. This will import all symbols from the other module. Additional symbol declarations are not allowed, because of tight coupling between a pattern and its symbols. Adding new symbols would imply an implicit not, as there is no way to add that symbol to the pattern. Event filters can also be reused with references to other event modules. This allows for advice sharing by using different patterns as pointcut. References have the following form: `concernname::modulename:identifier`.

These constructs allow for the specification of *abstract event modules*. An example is shown in Listing 6.5, where an abstract change pattern is defined and used by two event modules.

6.4 Conflicts between Event Modules

Having multiple event modules superimposed on the same object, might cause problems in the order of advice execution. Some modules might depend on other modules and might conflict with other modules. For filter modules these problems occur at shared join points. Event modules, on the other hand, execute their advice based on a certain trace against the program. Two modules might (partly) match on the same set of events, but execute their advice at distinct join points. If these two modules conflict with each other, this can cause unwanted side-effects. We will illustrate this with an example.

Suppose we refine the notify concern presented in Section 5.4.1 with the following notion. When adding or removing a list of products, notify all observers on every fifth update. The following pattern expression satisfies the given concern.

$$e_f((c_l r_l)^5)^+ \tag{6.1}$$

As before (in expression 5.1), symbol e_f represents events for execution of a method of class `Inventory`, and symbols c_l and r_l represent events for call and return of state changing methods to class `ArrayList`. Please note that symbol r_f is omitted. If we were to include it, the expression would never invoke its advice

```
concern abs_change_concern {
  eventmodule abs_change(??ls) {
    symbols
      Ef: before execution(?f:=selector);
      Rf: after execution(selector=?f);
      Cl: before call(?l:=selector:??ls, target=inner.stock);
      Rl: after call(selector=?l, target=inner.stock);
    pattern:
      change: {Ef (Cl Rl)+ Rf};
  }
}
```

(a) Abstract event module

```
concern inventory_change {
  eventmodule inventory_notify(??ls) {
    internals
      notifier: ExampleAOP.Notify;
    pattern
      abs_change_concern::abs_change:change;
    eventfilters
      notify: Meta = {notifier.Notify}
  }

  eventmodule inventory_logger(??ls) {
    externals
      logger: ExampleAOP.Logger = ExampleAOP.Logger.Instance();
    pattern
      abs_change_concern::abs_change:change;
    eventfilters
      logger: Meta = {logger.Log}
  }
}
```

(b) Two concrete event modules

Listing 6.5: Example demonstrating abstract event modules

on every fifth update. This exposes a potential conflict between the patterns of both event modules.

Suppose, for example, that at run-time `Inventory.AddProducts` is called with a list of ten products and both event modules are imposed on class `Inventory`. Expression 6.1 will match on the fifth and tenth return from a call to method `Add`, causing the notify advice to be invoked twice. When method `AddProducts` returns, expression 5.1 will also match and invoke the notify advice again. This behavior might only be desired when method `Add` was not called a multiple of five times. There are two approaches to remove such redundant advice execution. Either merge the two patterns into one or define a constraint model to specify exclusion conditions.

6.4.1 Merging Patterns

To resolve a conflict between patterns, a software developer can merge the conflicting patterns into one pattern. This does, however, require extra syntax constructs. The merged pattern, for our example, uses a counter to keep track of the number of calls to a state changing function c_l.

$$e_f(c_l r_l)^i r_f \tag{6.2}$$

There are two cases when the notify advice should be executed: if i modulo five equals zero; and when the pattern reaches a final state (on r_f) and i modulo five does not equal zero. But the current model only checks for advice execution when the automaton reaches an accepting state. We can add an extra operator to define extra accepting states in the pattern, or evaluate event filters on every transition. The extra operator would just clutter the pattern. Evaluating on every transition enables powerful new constructs. This requires every event filter to contain an appropriate condition which specifies if the associates action should be taken or not. Thus, a condition should reason about whether a state is accepting or failing, and what the value of the counters is.

Consider the example presented in Listing 6.6. Four conditions are defined which control the actions taken by event filters. Instead of defining the conditions in the concern, we could make an interface available to allow a developer to specify the condition in a programming language. This would keep the syntax for conditions simple. On every transition, all event filters will be evaluated. The first filter is an error filter, it disallows more than fifty changes per method execution. The second filter will notify when the pattern reaches an accepting state, but only if counter `i` is not a multiple of five. In that case, the third filter already invoked notify the last time.

Because of the similarities between the two conflicting patterns, a new pattern could be created. In general, this might not always be the case. It remains unclear if the extra semantics for the pattern language will be sufficient.

```
eventmodule InventoryUpdate {
  internals
    subject: Subject;
    policy: ChangePolicy;
  conditions
    c1: ?i > 0;
    c2: ?i > 50;
    c3: isAccepting();
    c4: ?i mod 5 == 0;
  pattern
    change: {Ef (Cl Rl)[?i] Rf};
  eventfilters
    f1: Error = {c2 => policy.TooMuch};
    f2: Invoke = {c1 & c3 & !c4 => subject.Notify};
    f3: Invoke = {c4 => subject.Notify}
}
```

Listing 6.6: Proposal for evaluation of filters on transitions

6.4.2 Constraint Model

A constraint model is proposed by Nagy [2006] for composing aspects at shared join points. It identified two requirements for composing aspects: ordering and conditional execution. Aspect ordering specifies the execution order of aspects at shared join points. Conditional execution specifies a condition for when the execution of an aspect depends on the outcome of other aspects. To satisfy these requirements the constraint model distinguishes between three main categories of constrainsts: ordering constraints, structural constraints, and control constraints. The model is a generic constraint model that can be built into various AOP languages. The overview presented here is focused on the integration with Compose⋆.

Ordering constraints specify the relative order of filter modules at a shared join point using the `before` statement.

Structural constraints specify which filter modules have to be, or cannot be, mutually present at a shared join point as designated by `include` and `exclude` statements.

Control constraints specify a sequential, conditional execution of filter modules, depending on the execution of related filter modules. This is done using the statements `if` (conditional), `skipif` (conditional skip), `ordif` (before with conditional), and `ordskipif` (before with conditional skip).

Listing 6.7 shows a superimposition section with examples for each constraint, where `x` and `y` are filter module identifiers and `r` is a return value. The return value is used as substitute value in case of a conflict. The semantics of the control constraints are as follows. In line 8, `x` is executed if execution of `y` succeeded. In line 9, execution of `x` is skipped if execution of `y` succeeded. In line 10, `x` is executed if execution of `y` succeeded with additional constraint `y` before `x`. In line 11, execution of `x` is skipped if execution of `y` succeeded with additional

```
superimposition {
  selectors
    s = {C | isClassWithName(C, 'ExampleAOP.Inventory')};
  constraints
    s <- x before y;
    s <- x includes y;
    s <- x excludes y;
    s <- x if y;
    s <- x skipif y with r;
    s <- x ordif y;
    s <- x ordskipif y with r;
}
```

Listing 6.7: Examples of superimposition with constraints

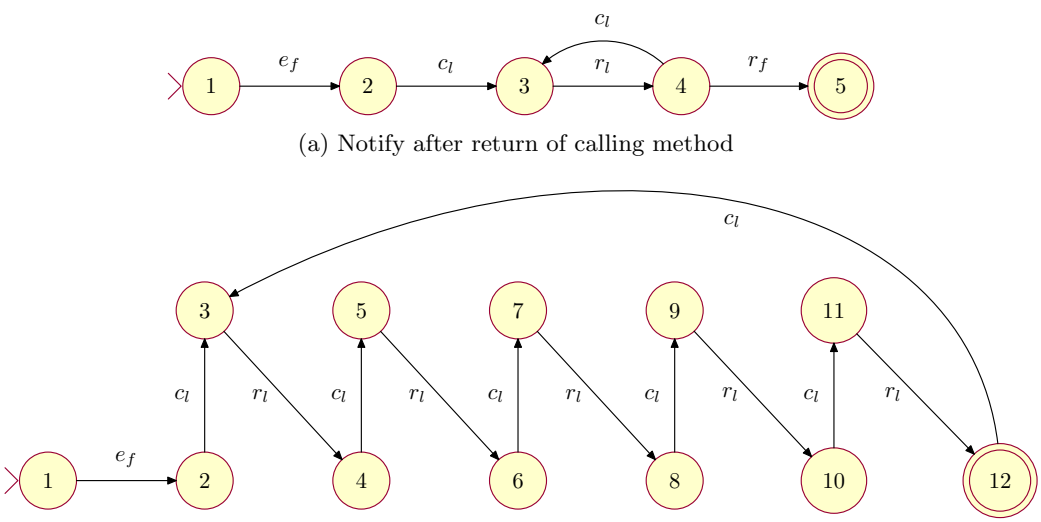
constraint `y` before `x`. The execution of a skipped filter module is substituted with return value `r`.

For a condition to determine whether the execution of a filter module succeeded, a default property called `isSucceeded` is added to the reified message. The value of that property has to be set explicitly by filters of a filter module. Whenever that filter module is used inside an `if` constraint, that property is checked.

Each of the constructs of the constraint model can also apply for event modules. But the model, however, is based on aspect composition at *shared join points*, while the problem discussed in the introduction of Section 6.4 takes place at two distinct events, namely r_l and r_f. Nonetheless, control constraints allow for conditional execution according to certain properties. These properties can be refined for event modules.

For event filter modules, an extra property `isAccepting` is added to indicate whether the pattern is in an accepting state or not. Its value is automatically set on each transition of an automaton. Whenever an event module is used inside an `if` constraint, both `isAccepting` and `isSucceeded` are checked. If a pattern ends with a recursive operation (i.e., Kleene star or plus), an accepting state can be reached more than once. If a pattern contains multiple recursive operations it might even have more than one accepting state. We can define a constraint relation which evaluates whether the current state is accepting or not by using constraint `x skipif y with void`, where `x` is the event module identifier for expression 5.1 and `y` for expression 6.1. On the fifth and tenth return to call `Add`, expression 6.1 reaches an accepting state, thus property `isAccepting` is true. If the notify advice executes successfully, property `isSucceeded` is also true. If the next event is returning from method `AddProducts`, constraint `skipif` will prevent a double notify because both properties are true. If another call to `Add` occured in between, expression 6.1 would not be in an accepting state anymore. The matching process can be visualized by doing the state transitions in Figure 6.1a and 6.1b. To not clutter the automata, *skip* and $fail$ transitions have been omitted.

Please note, that the semantics of the implicit fail model have consequences for control constraints. When declaring more symbols than used by a pattern



(a) Notify after return of calling method

(b) Notify every fifth update

Figure 6.1: Finite automata for notifications

might cause an automaton to fail. If symbol r_f was declared in the event module for notifying every fifth update, the automaton given in Figure 6.1b would be in the $fail$ state when the automaton of Figure 6.1a reaches its accepting state. In which case the constraint will not prevent a double notify. Unfortunately this is exactly where the problem lies in our example.

If a pattern expression does not contain a closure, the associated automaton is likely to never terminate. In our example there might always be another call and return to a state changing function (events c_l and r_l). When these events do not occur, the automaton will stay in its current state until the base program exits. A function execution to class `Inventory` (e_f) might terminate the pattern, but only if that event has the same properties as the initial event. In any case, every new function execution starts a new pattern.

When we fix expression 6.1 by adding an implicit or explicit ignore on r_f, extra properties are needed for the conflict relation. For clarity, Figure 6.2 depicts the total automaton for notification on every second update. When event r_f occurs, the fail state is reached and the automaton terminates. If the previous state was an accepting state, no updates have taken place after notification. In this case the conflict relation should prevent another update. By adding the following properties `isFailing`, `previousAccepted`, next to `isAccepting`, more complex constraint relations can be formulated. The constraint becomes `x skipif y.isFailing() and y.previousAccepted()with void`.

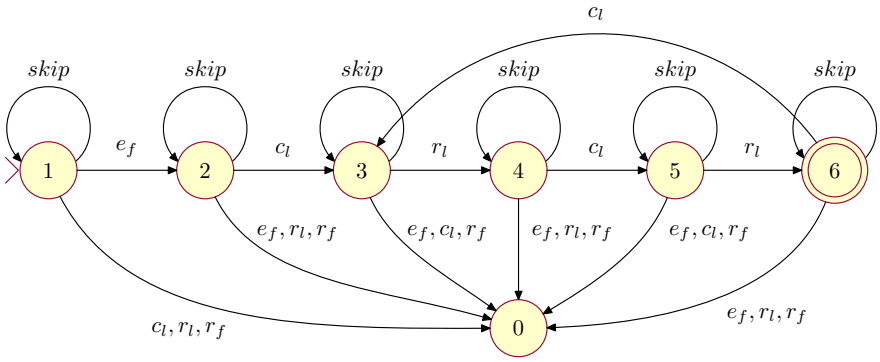


Figure 6.2: Finite automata for notifications every second update

6.4.3 Discussion

We presented two appoaches for resolving conflicts. The first approach increased the expressiveness of our language by evaluating filters on every transition and coupling them with more advanced conditions. The downside is that the developer needs to merge the conflicting patterns into a new pattern. This might not always be possible in a straightforward manner. The added expressiveness complicates the pattern language for corner cases.

The constraint model assumes the conflict is at a higher abstraction level. When using a constraint model there is no need to merge the patterns themselves. To resolve the conflict, expressiveness is increased through a set of abstract constraints, which enables patterns stay independent of each other. Because superimposition specifies which event modules are bound to which objects, it is a logical place to resolve conflicts between event modules too.

CHAPTER 7

Design

The architecture of Compose⋆, as described in Section 2.4, consists of four parts: an IDE interface, a compile-time, an adaptation layer, and a run-time. To implement events in the architecture of Compose⋆, changes need to be made to the last three parts.

The *compile-time* part contains the platform independent code for parsing concerns, and writing relevant information to a repository and a weave specification. The repository specifies the binding between modules in compile-time and run-time. This chapter describes how event modules are parsed and stored in the repository, and how automata are created for each pattern. The classes on which event modules are superimposed, need to be instrumented for each defined symbol. This is specified in a weave specification. That specification is used during *adaptation* to weave the hooks in the target program. At *run-time* each hook sends out an event to an event monitor associated to an object. The event monitor will keep state for each pattern imposed on that object and invoke advice when needed. The run-time matching process is discussed in the last section.

7.1 Event Module EBNF Grammar

Backus-Naur form (BNF) is a formal mathematical way to describe the grammar of a language. We use extended BNF (EBNF) notation to specify the grammar of event modules in Compose⋆.

Each rule of the grammar describes a non-terminal symbol. The defining symbol in a rule is '::='. In the right-hand side of a rule, the following elements of EBNF can be used:

[]	Specifies an element that is optional;
()∗	Specifies an element that can be repeated zero or more times;
()+	Specifies an element that can be repeated one to more times;
' '	Specifies a string literal;
⟨*a*⟩ | ⟨*b*⟩	Specifies an alternative to a rule;
⟨*a-LIST*⟩	This expression is substituted with ⟨*a*⟩ ('`,`' ⟨*a*⟩)∗
⟨*a-SEQ*⟩	This expression is substituted with ⟨*a*⟩ ('`;`' ⟨*a*⟩)∗

Only the necessary modifications for inclusion of event modules to the grammar of Compose⋆ are specified. For the full specification, see the Compose⋆ grammar on the documentation page of the Compose⋆ website[1].

Concern Definition

An event module block is added to the concern definition. A concern definition can consist of zero or more event modules.

```
⟨Concern⟩          ::= 'concern' ⟨ConcernName⟩
                       ['('⟨FormalConcernParameter-SEQ⟩')']
                       ['in' ⟨PackageReference⟩] '{'
                       (⟨FilterModule⟩)* (⟨EventModule⟩)* [⟨Superimposition⟩]
                       [⟨Implementation⟩] '}'
```

Event Module Definition

An event module can consists of internals, externals, conditions, symbols, a pattern and event filters. Internals, externals and conditions have the same definition as used in filter modules.

```
⟨EventModule⟩                  ::= 'eventmodule' ⟨EventModuleName⟩
                                   ['('⟨FormalModuleParameter-LIST⟩')'] '{'
                                   [⟨Internals⟩] [⟨Externals⟩] [⟨Conditions⟩]
                                   [⟨Symbols⟩] [⟨EventPattern⟩] [⟨EventFilters⟩] '}'
⟨Internals⟩                    ::= 'internals' (⟨Identifier-LIST⟩ ':' ⟨Type⟩ ';')*
⟨Externals⟩                    ::= 'externals' (⟨Identifier-LIST⟩ ':' ⟨Type⟩
                                   ['=' ⟨InitializationExpression⟩] ';')*
⟨InitializationExpression⟩     ::= ⟨ModuleElementReference⟩ | ⟨FullQualifiedName⟩
⟨Conditions⟩                   ::= 'conditions' (⟨ConditionDeclaration⟩)*
⟨ConditionDeclaration⟩         ::= ⟨ConditionName⟩ ':' ⟨ConditionReference⟩ ';'
⟨Symbols⟩                      ::= 'symbols' (⟨SymbolDeclaration⟩)+
⟨EventPattern⟩                 ::= 'pattern' (⟨PatternDeclaration⟩ | ⟨PatternReference⟩)
⟨EventFilters⟩                 ::= 'eventfilters' ⟨GeneralEventFilterSet⟩
```

[1] See `ComposestarDocumentation` on http://composestar.sourceforge.net/.

Symbol Definition

Symbols declare which identifiers can be used as literal inside pattern definitions. They consist of an event type and optional properties to filter on specific events.

⟨*SymbolDeclaration*⟩	::=	⟨*SymbolName*⟩ '`:`' ⟨*EventModifier*⟩ ⟨*EventType*⟩ ['`(`' ⟨*EventPropertyExpression-LIST*⟩ '`)`'] '`;`'
⟨*EventModifier*⟩	::=	'`before`' | '`after`'
⟨*EventPropertyExpression*⟩	::=	⟨*EventProperty*⟩ [⟨*EventPropertyOperator*⟩ ⟨*EventProperty*⟩]
⟨*EventProperty*⟩	::=	⟨*EventPropertyName*⟩ ['`:`' ⟨*ListParameterName*⟩]
	|	⟨*ParameterName*⟩
⟨*EventPropertyOperator*⟩	::=	'`:=`' | '`=`' | '`!=`'
⟨*EventPropertyName*⟩	::=	⟨*Identifier*⟩

Event Pattern Definition

A pattern consists of an expression and a optional condition. The precedence between operators is as follows:

- Quantifiers (Kleene star, Kleene plus, repetition, and optional composition);
- Sequential composition;
- Alternative composition.

Grouping overrides precedence. For clarity, precedence is not shown in this EBNF.

⟨*PatternDeclaration*⟩	::=	⟨*PatternName*⟩ '`:`' '`{`' ⟨*PatternExpression*⟩ ['`|`' ⟨*PatternCondition-LIST*⟩] '`}`' '`;`'
⟨*PatternExpression*⟩	::=	⟨*PatternSymbol*⟩
	|	⟨*PatternGrouping*⟩
	|	⟨*PatternKleeneStar*⟩
	|	⟨*PatternKleenePlus*⟩
	|	⟨*PatternRepetition*⟩
	|	⟨*PatternOptional*⟩
	|	⟨*PatternSequence*⟩
	|	⟨*PatternAlternative*⟩

```
⟨PatternSymbol⟩                ::= ⟨SymbolName⟩
⟨PatternGrouping⟩              ::= '(' ⟨PatternExpression⟩ ')'
⟨PatternSequence⟩              ::= ⟨PatternExpression⟩ (' ' | '.') ⟨PatternExpression⟩
⟨PatternAlternative⟩           ::= ⟨PatternExpression⟩ '|' ⟨PatternExpression⟩
⟨PatternOptional⟩              ::= ⟨PatternExpression⟩ '?'
⟨PatternKleeneStar⟩            ::= ⟨PatternExpression⟩ '*'
⟨PatternKleenePlus⟩            ::= ⟨PatternExpression⟩ '+'
⟨PatternRepetition⟩            ::= ⟨PatternExpression⟩
                                   '[' (⟨Number⟩ | ⟨PatternCounterName⟩) ']'
⟨PatternCondition⟩             ::= ⟨PatternConditionGreater⟩
                                 | ⟨PatternConditionGreaterEqual⟩
⟨PatternConditionGreater⟩      ::= ⟨PatternCounterName⟩ '>' ⟨NaturalNumber⟩
⟨PatternConditionGreaterEqual⟩ ::= ⟨PatternCounterName⟩ '>=' ⟨NaturalNumber⟩
⟨PatternCounterName⟩           ::= '?' ⟨Identifier⟩
```

Event Filter Definitions

Event filters have a lot in common with message filters. They consist of a condition part and a matching part. A substitution part is specific to message filters. The target of a message can be specified in terms of pseudo-variables [Bergmans, 1994]. Variable `inner` refers to the implementation part of the current object. Variable `self` refers to the object that executes the current method. Variable `sender` refers to the object that was responsible for sending the currently executed message. Variable `server` refers to the original receiver of the message that caused the current execution.

```
⟨GeneralEventFilterSet⟩        ::= ⟨GeneralEventFilter⟩
                                   (⟨FilterCompositionOperator⟩ ⟨GeneralEventFilter⟩)*
⟨FilterCompositionOperator⟩    ::= ';'
⟨GeneralEventFilter⟩           ::= ⟨EventFilterName⟩ ':' ⟨EventFilterType⟩
                                   ['(' ⟨ActualFilterArguments⟩ ')'] '='
                                   '{' ⟨EventFilterElements⟩ '}'
⟨ActualFilterParameters⟩       ::= ⟨Value-LIST⟩
⟨EventFilterElements⟩          ::= ⟨EventFilterElement⟩
                                   (⟨ElementCompositionOperator⟩ ⟨EventFilterElement⟩)*
⟨ElementCompositionOperator⟩   ::= ','
⟨EventFilterElement⟩           ::= [⟨ConditionExpression⟩ ⟨ConditionOperator⟩]
                                   [⟨EventTarget⟩ '.'] ⟨Selector⟩
⟨EventTarget⟩                  ::= ⟨Identifier⟩ | 'inner' | 'self' | 'sender' | 'server'
```

```
⟨ConditionOperator⟩      ::= '=>' | '~>'

⟨ConditionExpression⟩    ::= ⟨ConditionLiteral⟩
                           | ⟨ConditionGrouping⟩
                           | ⟨ConditionNot⟩
                           | ⟨ConditionAnd⟩
                           | ⟨ConditionOr⟩

⟨ConditionGrouping⟩      ::= '(' ⟨ConditionExpression⟩ ')'

⟨ConditionNot⟩           ::= '!' ⟨ConditionExpression⟩

⟨ConditionAnd⟩           ::= ⟨ConditionExpression⟩ '&' ⟨ConditionExpression⟩

⟨ConditionOr⟩            ::= ⟨ConditionExpression⟩ '|' ⟨ConditionExpression⟩

⟨ConditionLiteral⟩       ::= ⟨ConditionName⟩ | 'True' | 'False'
```

Superimposition Definition

An event module binding part is added to also superimpose event modules on objects.

```
⟨Superimposition⟩        ::= 'superimposition' '{'
                             [⟨SelectorDefinition⟩]
                             [⟨ConditionBinding⟩]
                             [⟨FilterModuleBinding⟩]
                             [⟨EventModuleBinding⟩]
                             [⟨AnnotationBinding⟩]
                             [⟨Constraints⟩] '}'
```

Event Module Binding Definition

The event module binding describes on which objects, as identified by a selector, event modules are superimposed.

```
⟨EventModuleBinding⟩     ::= 'eventmodules'
                             (⟨CommonBindingPart⟩ ⟨EventModuleSet⟩ ';')*

⟨EventModuleSet⟩         ::= '{' ⟨ConcernElementReference-LIST⟩ '}'
                           | ⟨ConcernElementReference-LIST⟩

⟨CommonBindingPart⟩      ::= ⟨Selector⟩ ⟨WeaveOperator⟩

⟨Selector⟩               ::= ⟨ConcernElementReference⟩

⟨WeaveOperator⟩          ::= '<-'
```

References

References enable concern elements to refer to each other.

⟨*PackageName*⟩	::=	⟨*Identifier*⟩
⟨*PackageReference*⟩	::=	(⟨*PackageName*⟩ '`.`')* ⟨*PackageName*⟩
⟨*ConcernName*⟩	::=	⟨*Identifier*⟩
⟨*FormalConcernParameter*⟩	::=	⟨*Identifier-LIST*⟩ '`:`' ⟨*Type*⟩
⟨*ConcernReference*⟩	::=	[⟨*PackageReference*⟩ '`.`'] ⟨*ConcernName*⟩
⟨*ConcernElementReference*⟩	::=	[⟨*ConcernReference*⟩ '`::`'] ⟨*Identifier*⟩
⟨*ModuleElementReference*⟩	::=	[[⟨*ConcernReference*⟩ '`::`'] ⟨*ModuleName*⟩ '`:`'] ⟨*Identifier*⟩
⟨*ModuleName*⟩	::=	⟨*FilterModuleName*⟩ | ⟨*EventModuleName*⟩
⟨*EventModuleName*⟩	::=	⟨*Identifier*⟩
⟨*FormalModuleParameter*⟩	::=	⟨*ParameterName*⟩ | ⟨*ListParameterName*⟩
⟨*ParameterName*⟩	::=	'`?`' ⟨*Identifier*⟩
⟨*ListParameterName*⟩	::=	'`??`' ⟨*Identifier*⟩
⟨*ConditionName*⟩	::=	⟨*Identifier*⟩
⟨*SymbolName*⟩	::=	⟨*Identifier*⟩
⟨*PatternName*⟩	::=	⟨*Identifier*⟩
⟨*PatternReference*⟩	::=	[⟨*ConcernReference*⟩ '`::`'] ⟨*EventModuleName*⟩ '`:`' ⟨*PatternName*⟩
⟨*EventFilterName*⟩	::=	⟨*Identifier*⟩
⟨*Type*⟩	::=	⟨*ConcernReference*⟩
⟨*Value*⟩	::=	⟨*Identifier*⟩ | ⟨*Number*⟩

General Definitions

General definitions of which characters can be used for each construct in a concern specification.

⟨*Identifier*⟩	::=	(⟨*Letter*⟩ | ⟨*Special*⟩)(⟨*Letter*⟩ | ⟨*Digit*⟩ | ⟨*Special*⟩)*
⟨*Number*⟩	::=	['`-`'] (⟨*Digit*⟩)+
⟨*NaturalNumber*⟩	::=	(⟨*Digit*⟩)+
⟨*Letter*⟩	::=	⟨*UpperCase*⟩ | ⟨*LowerCase*⟩
⟨*UpperCase*⟩	::=	'`A`' | '`B`' | .. | '`Z`'
⟨*LowerCase*⟩	::=	'`a`' | '`b`' | .. | '`z`'
⟨*Digit*⟩	::=	'`0`' | '`1`' | .. | '`9`'
⟨*Special*⟩	::=	'`_`'

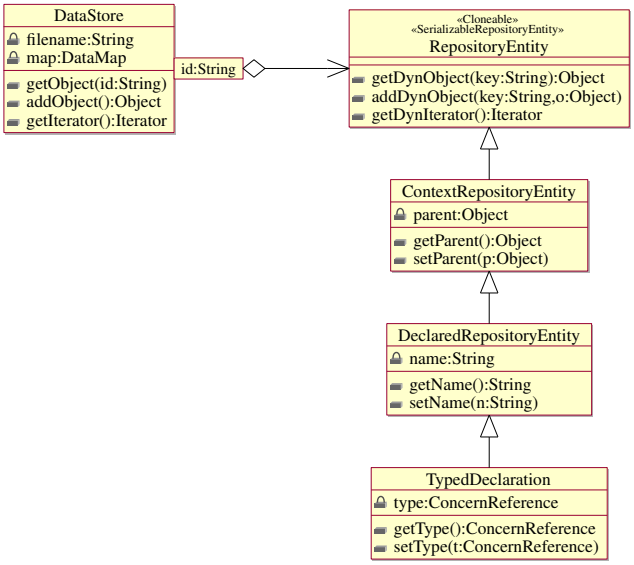


Figure 7.1: Simplified class diagram of repository

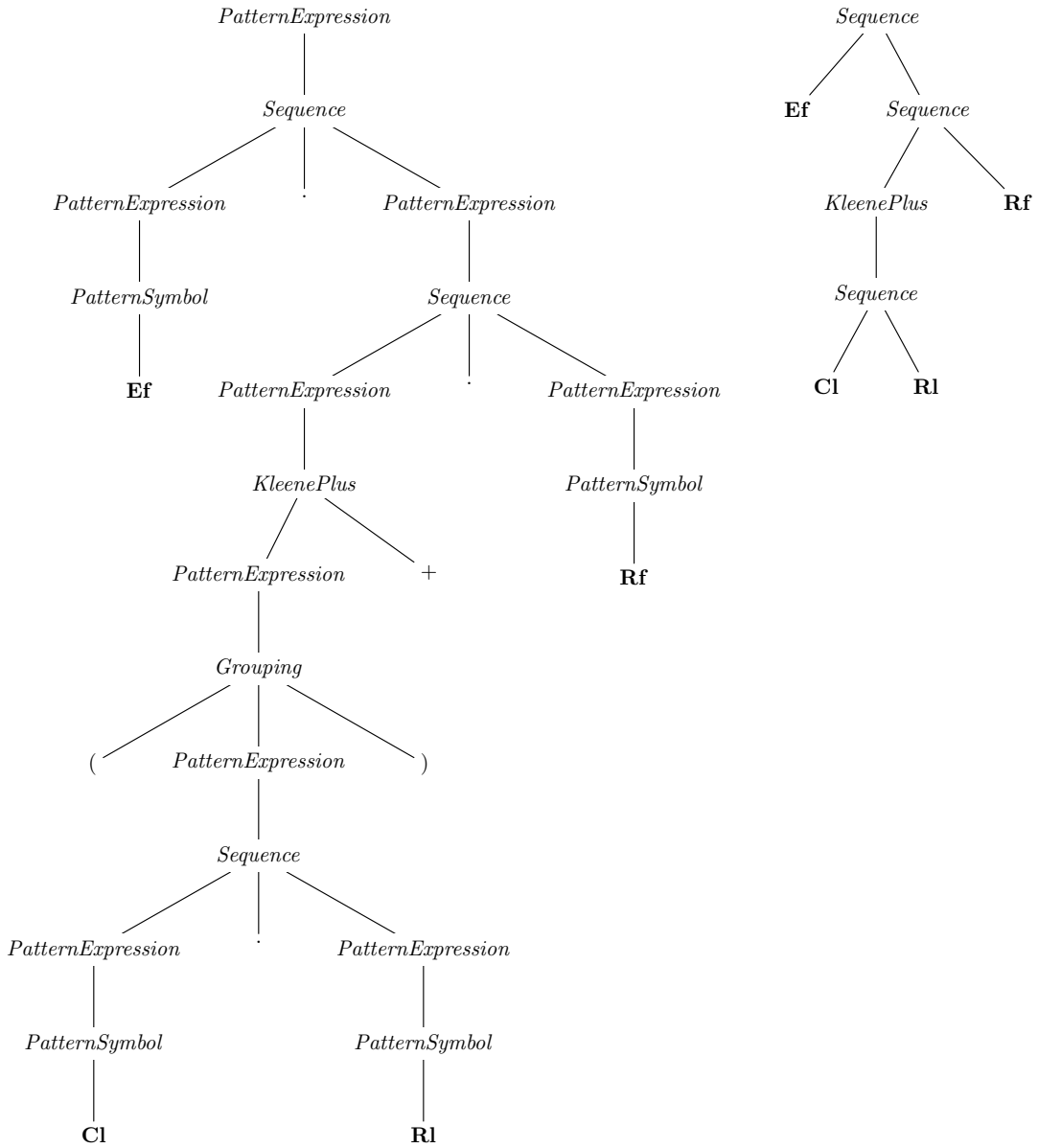
7.2 Repository

The repository is used by different modules in Compose⋆ to share data. It is implemented as a data store which contains a map of objects. All objects are inserted with a unique key. They can be retrieved by that key or through an iterator. A simplified version of the class diagram is shown in Figure 7.1.

All objects stored in the repository must inherit from class `RepositoryEntity`. Class `ContextRepositoryEntity` provides a parent link to store entities as children of other entities. Class `DeclaredRepositoryEntity` provides a qualified name for objects. Used for storing filter modules, for example. Class `TypedDeclaration` provides a concern reference to store a pointer to a concern. Used for storing internals and externals, for example. For more information on the repository, see M. Sc. thesis of Vinkes [2004].

7.3 Parsing Concerns for Events Modules

The compile-time of Compose⋆ uses ANTLR (ANother Tool for Language Recognition) for parsing concern specifications. To support parsing of event modules, the grammar presented in Section 7.1 is slightly modified to remove left recursion and to incorporate operator precedence. This is needed to eliminate ambiguous derivations. ANTLR uses the grammar to generate a parser. By parsing a concern specification, a parse tree is created. A parse tree, also called *concrete syntax tree* (CST), depicts how a concern can be derived from the start symbol by applying production rules found in the grammar. Every branch node in that tree is a nonterminal and corresponds to the left-hand side of a production rule. An example of a parse tree is shown in Figure 7.2a. It depicts a partial parse tree, consisting of the pattern expression specified in concern `InventoryUpdates`



(a) Concrete syntax tree (b) Abstract syntax tree

Figure 7.2: Parse trees of the change pattern from Listing 6.2

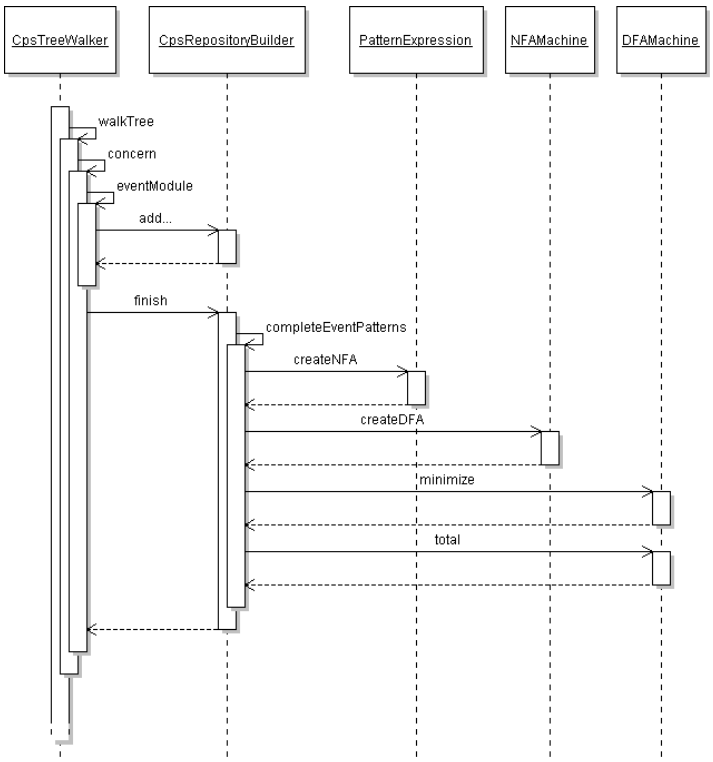


Figure 7.3: Sequence diagram for creating event modules during compile-time

of Listing 6.2. From such a parse tree ANTLR creates an *abstract syntax tree* (AST) by leaving out irrelevant nonterminals, like `PatternExpression`, `Grouping`, and `Symbol` for example. Operators and other elements which need to be available in the repository are kept. The AST, corresponding to the given CST, is depicted in Figure 7.2b. In that example, grouping is not an operator, it is only used to steer the derivation process to override precedence between operators, and is thus removed in the AST.

Information from the AST is visited by a tree walker which invokes the appropriate method on an instance of class `CpsRepositoryBuilder` to generate objects stored in the repository. This includes translation of syntactic sugar by expanding or adding repository entries. When all event module objects are build, method `finish` initiates the construction of all automaton. A sequence diagram is depicted in Figure 7.3. Classes `PatternExpression`, `NFAMachine`, and `DFAMachine` are explained in the next sections.

To extend the repository for storing event modules, new entities are defined. Figure 7.4 shows a class diagram of repository elements which are declared by name. Declared by name means that another element in the repository is able to reference them. For event modules they are declared symbols, a pattern, event filters, conditions, internals and externals. Event modules can have parameters, they are stored in a dictionary and implemented like filter module parameters as explained by Doornenbal [2006]. To support multiple matching processes on a pattern, structure and state are separated. The pattern structure is refer-

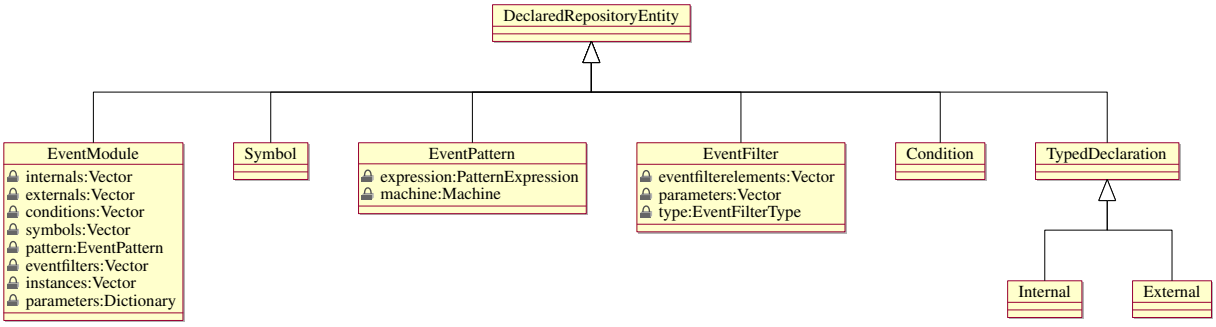


Figure 7.4: Class diagram of declared repository elements for event modules

enced through class `EventPattern` and the state of a matching process is stored in a vector of instances. Event filters have a type, parameters and consist of `EventFilterElements`. The repository entities for conditions, internals and externals are the same entities as used in filter modules.

Unlike declared entities, context entities are not referred to by name. They are intended for elements that are part of other entities. Examples are filter elements, condition expressions, and pattern expressions, which are part of the declared entities `EventModule` and `EventPattern`. The context entities needed for event modules are depicted in Figure 7.5. Instances of class `EventFilterElement` make up the parts of an event filter. Depending on the condition, it can send a message to a target and selector. Multiple event filter elements can be specified if there is a right operator present. Class `EventModuleOrder` defines the ordering relation between event modules. Its implementation is analogous to `FilterModuleOrder`. Class `ConditionExpression` contains a composite structure of conditions. Its implementation is shared by message filters. Class `PatternExpression` contains a composite structure to store the pattern expression given in the concern specification. The other entities are used for storage of the state machine of an event module. They are explained in Section 7.4.

The third and last part needed for storing event modules in the repository, is storing references to declared concern elements that are part of event modules. The class diagram is shown in Figure 7.6. Each reference contains a boolean flag whether the name is resolved or not. Class `DeclaredObjectReference` is used to refer to internals and externals.

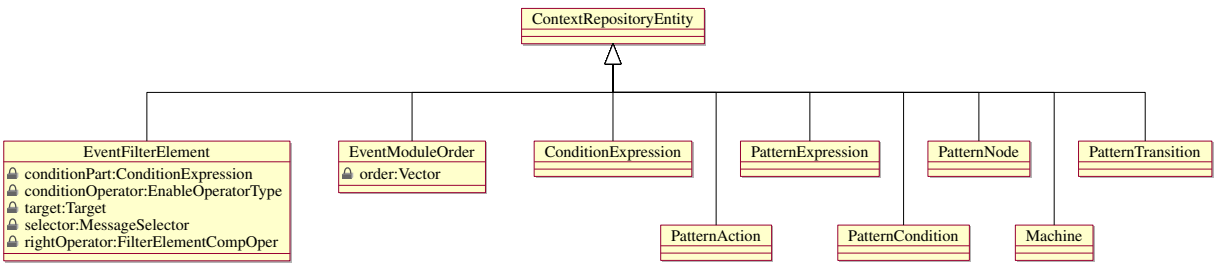


Figure 7.5: Class diagram of context repository elements for event modules

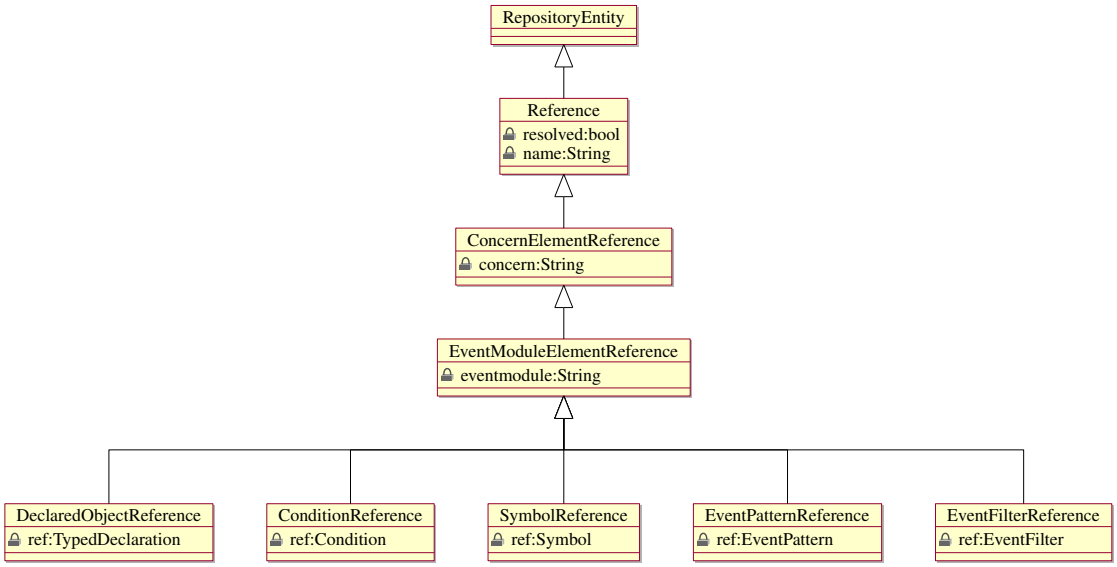


Figure 7.6: Class diagram for storing references to declared elements of event modules

7.4 Generation of Automata

In Section 5.4.3 we explained that patterns can be represented as automata. In this section we show how the compile-time part of Compose★ builds a deterministic finite automaton (DFA) from an expression through intermediate steps. The first step is generating a nondeterministic automaton from a pattern expression. That automaton is made deterministic and minimized.

7.4.1 Nondeterministic Finite Automaton

When a concern specification is parsed, the resulting AST for a pattern expression is build up from instances of the classes shown in Figure 7.7. An example of such a partial AST was given in Figure 7.2b. Class `PatternExpression` is implemented using the composite design pattern [Gamma et al., 1995] and serves as base class for pattern symbols and operators. An AST is used to construct a nondeterministic finite automaton (NFA) by calling method `createNFA` on the root node. It builds an NFA using *Thompson's construction* algorithm [Giammarresi et al., 2004].

An NFA is a state machine where, based on the current state and input symbol, there may be several possible next states. Besides nondeterminism, we allow transitions that do not consume an input symbol. They are called *epsilon* (ϵ) transitions. Nondeterminism and ϵ-transitions make state machines easier to construct. Nondeterminism allows for compact 'or' constructs between states and ϵ-transitions allow 'jumping' between states, used by recursion for example. These extra constructs, however, do not add expressive power. Any ϵ-NFA can be converted to an equivalent DFA.

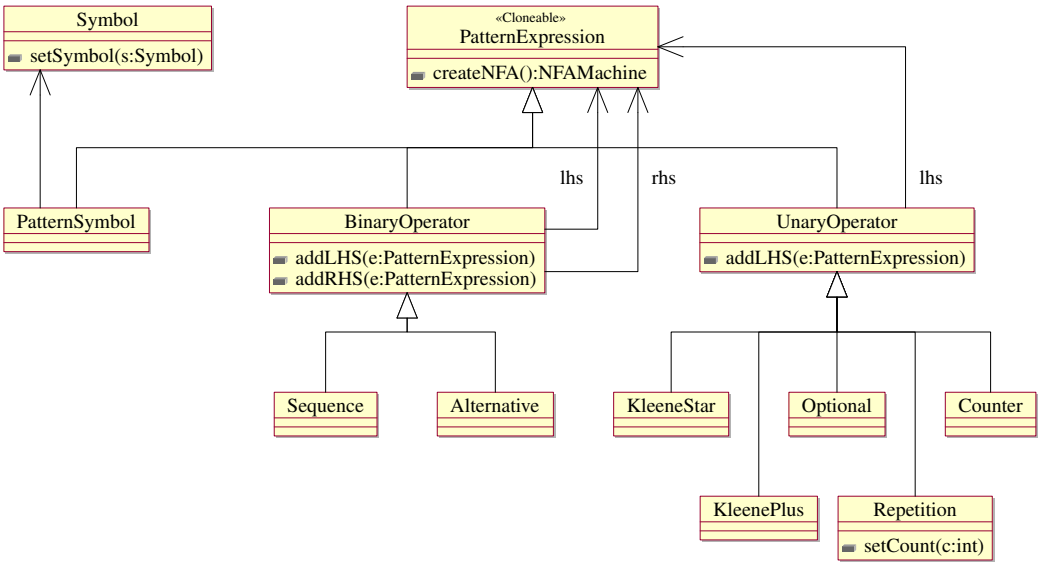


Figure 7.7: Class diagram of pattern expression for AST

States in a state machine are represented by instances of class `PatternNode`. Each node contains a set of transitions coupled with a symbol. A node can be accepting or not. A node is marked failed, if it is not accepting and does not have any transitions to other nodes. If an input string does not match a pattern, the automaton reaches such a failed state. The class diagram is shown in Figure 7.8.

In a depth-first traversal, an ϵ-NFA is constructed by invoking abstract method `createNFA` on each node (branch and leaf) of the AST. Method `createNFA` is overridden in derived classes to construct states (instances of `NFAPatternNode`) and create transitions between them. It returns an instance of `NFAMachine`, which contains a reference to the initial state and the final state of the constructed automaton. These references are used to alter the automaton during composition. Because of how we compose an automaton from basic elements, each

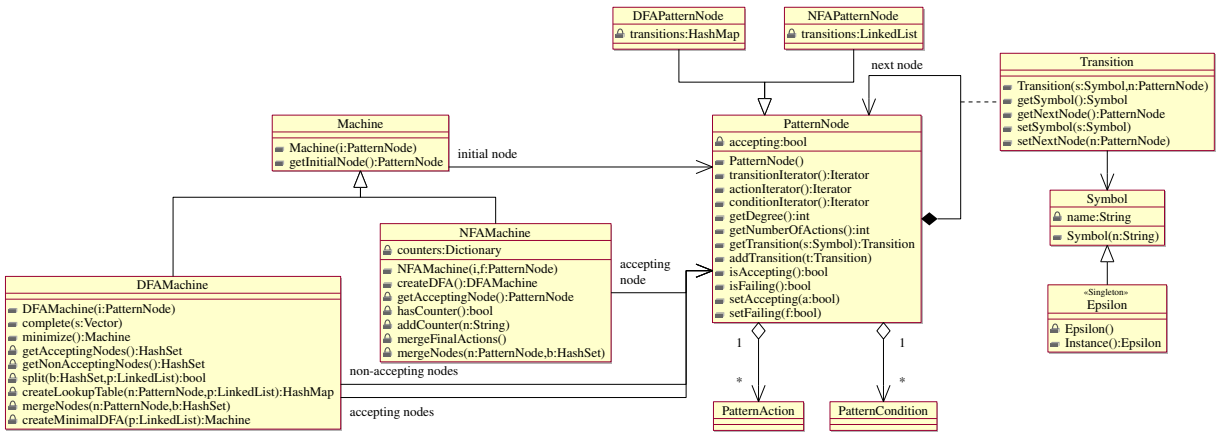


Figure 7.8: Class diagram of state machine

ϵ-NFA has only one final state. Every class that extends from abstract class `PatternExpression` should implement interface `Cloneable`. It is used to define complex operators in terms of other operators. Examples of basic operators are sequential composition, alternative composition, and Kleene star. An example of a complex operator is repetition. Class `PatternNode` is an abstract class to define a generic interface to access a node (a state in a machine). It is implemented as strategy pattern [Gamma et al., 1995]. A nondeterministic automaton can have multiple transitions on the same symbol, thus class `NFAPatternNode` contains a linked list of transitions. A deterministic automaton can only have one transition for a certain symbol, thus class `DFAPatternNode` uses a hashed map of transitions.

Pattern Symbols

A pattern symbol creates two nodes (states) with one transition between them. The first node is the initial node, the last node is the final node. This is the machine returned by method `createNFA` of class `PatternSymbol`. Below is an example of two machines consisting of only one transition.

When parsing a concern specification, all declared symbols are stored in a dictionary inside an event module. A symbol name in the pattern expression corresponds to one of those objects. The symbol name is looked up in the dictionary and passed as argument to the constructor of class `PatternSymbol`. Method `createNFA` constructs two new nodes, adds a transition, and returns the resulting machine. The implementation in Java is shown in Listing 7.1.

Pattern Operators

Binary operators, like sequence and alternative, operate on a left-hand side and a right-hand side. Listing 7.2 shows the abstract class for binary operators. It is responsible for the composite structure of pattern expression, or child management operations as Gamma et al. [1995] call it in the composite design pattern. Unary operators are defined in a similar manner, but they only operate on a left-hand side.

```
public class PatternSymbol extends PatternExpression
{
  private Symbol symbol;

  public void setSymbol(Symbol s) {
    symbol = s;
  }

  public Object clone() {
    PatternSymbol copy_self = (PatternSymbol)super.clone();
    copy_self.setSymbol(symbol);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAPatternNode initialNode = new NFAPatternNode();
    NFAPatternNode finalNode = new NFAPatternNode();
    finalNode.setAccepting(true);
    initial.addTransition(new Transition(symbol, finalNode));
    return new NFAMachine(initialNode, finalNode);
  }
}
```

Listing 7.1: Implementation of expressions with one symbol

```
public abstract class BinaryOperator extends PatternExpression
{
  private PatternExpression lhs, rhs;

  public void addLHS(PatternExpression p) {
    lhs = p;
  }

  public void addRHS(PatternExpression p) {
    rhs = p;
  }
}
```

Listing 7.2: Abstract class for binary operators in an expression

```
public class Sequence extends BinaryOperator
{
  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.clone();
    PatternExpression copy_rhs = (PatternExpression)rhs.clone();
    Sequence copy_self = (Sequence)super.clone();
    copy_self.addLHS(copy_lhs);
    copy_self.addRHS(copy_rhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAMachine m1 = lhs.createNFA();
    NFAMachine m2 = rhs.createNFA();
    NFAPatternNode m1_initial = m1.getInitialNode();
    NFAPatternNode m1_final = m1.getAcceptingNode();
    NFAPatternNode m2_initial = m2.getInitialNode();
    NFAPatternNode m2_final = m2.getAcceptingNode();
    m1_final.setAccepting(false);
    Symbol epsilon = (Symbol)Epsilon.Instance();
    m1_final.addTransition(new Transition(epsilon, m2_initial));
    return new NFAMachine(m1_initial, m2_final);
  }
}
```

Listing 7.3: Implementation of sequential composition operator

Sequential Composition

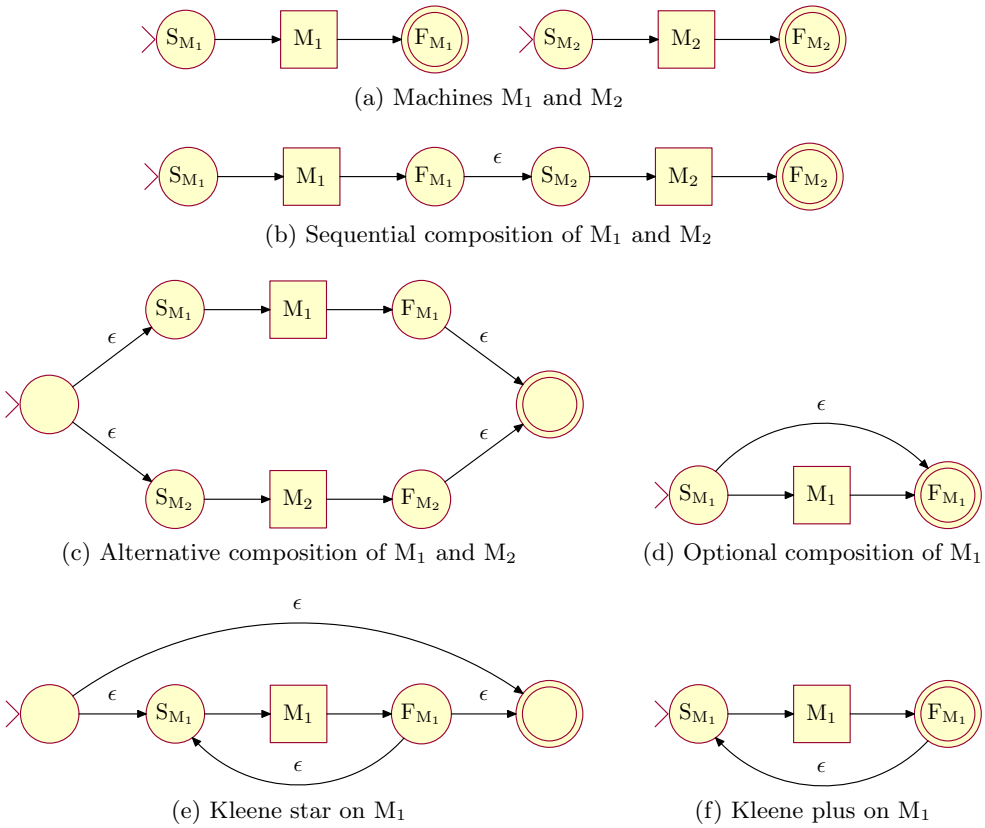
Figure 7.9b depicts the composition of two machines M_1 and M_2 of Figure 7.9a. The final node of the first machine is connected to the initial node of the second machine by an ϵ-transition. Class `Epsilon` is implemented as singleton, which enables it to be reused for all ϵ-transitions. Listing 7.3 shows the implementation of the sequence operator.

Alternative Composition

Alternative composition of two machines M_1 and M_2 is depicted in Figure 7.9c. New initial and final nodes are added with ϵ-transitions to initial and final states of both machines. This is implemented as shown in Listing 7.4.

Optional Composition

Figure 7.9d depicts how machine M_1 is made optional. Optional composition is a unary operator which adds an ϵ-transition from the initial state of M_1 to its final state. Listing 7.5 shows the implementation of the optional composition operator.



(a) Machines M_1 and M_2

(b) Sequential composition of M_1 and M_2

(c) Alternative composition of M_1 and M_2

(d) Optional composition of M_1

(e) Kleene star on M_1

(f) Kleene plus on M_1

Figure 7.9: Composition mechanisms for construction of ϵ-NFA

```
public class Alternative extends BinaryOperator
{
  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.clone();
    PatternExpression copy_rhs = (PatternExpression)rhs.clone();
    Alternative copy_self = (Alternative)super.clone();
    copy_self.addLHS(copy_lhs);
    copy_self.addRHS(copy_rhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAMachine m1 = lhs.createNFA();
    NFAMachine m2 = rhs.createNFA();
    NFAPatternNode m1_initial = m1.getInitialNode();
    NFAPatternNode m1_final = m1.getAcceptingNode();
    NFAPatternNode m2_initial = m2.getInitialNode();
    NFAPatternNode m2_final = m2.getAcceptingNode();
    m1_final.setAccepting(false);
    m2_final.setAccepting(false);
    Symbol epsilon = (Symbol)Epsilon.Instance();
    NFAPatternNode initialNode = new NFAPatternNode();
    NFAPatternNode finalNode = new NFAPatternNode();
    finalNode.setAccepting(true);
    initialNode.addTransition(new Transition(epsilon, m1_initial));
    initialNode.addTransition(new Transition(epsilon, m2_initial));
    m1_final.addTransition(new Transition(epsilon, finalNode));
    m2_final.addTransition(new Transition(epsilon, finalNode));
    return new NFAMachine(initialNode, finalNode);
  }
}
```

Listing 7.4: Implementation of alternative composition operator

```
public class Optional extends UnaryOperator
{
  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.clone();
    Optional copy_self = (Optional)super.clone();
    copy_self.addLHS(copy_lhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAMachine m1 = lhs.createNFA();
    NFAPatternNode m1_initial = m1.getInitialNode();
    NFAPatternNode m1_final = m1.getAcceptingNode();
    Symbol epsilon = (Symbol)Epsilon.Instance();
    m1_initial.addTransition(new Transition(epsilon, m1_final));
    return new NFAMachine(m1_initial, m1_final);
  }
}
```

Listing 7.5: Implementation of optional composition operator

```
public class KleeneStar extends UnaryOperator
{
  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.Clone();
    KleeneStar copy_self = (KleeneStar)super.clone();
    copy_self.addLHS(copy_lhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAMachine m1 = lhs.createNFA();
    NFAPatternNode m1_initial = m1.getInitialNode();
    NFAPatternNode m1_final = m1.getAcceptingNode();
    m1_final.setAccepting(false);
    Symbol epsilon = (Symbol)Epsilon.Instance();
    NFAPatternNode initialNode = new NFAPatternNode();
    NFAPatternNode finalNode = new NFAPatternNode();
    finalNode.setAccepting(true);
    initialNode.addTransition(new Transition(epsilon, m1_initial));
    initialNode.addTransition(new Transition(epsilon, finalNode));
    m1_final.addTransition(new Transition(epsilon, finalNode));
    m1_final.addTransition(new Transition(epsilon, m1_initial));
    return new NFAMachine(initialNode, finalNode);
  }
}
```

Listing 7.6: Implementation of Kleene star operator

Kleene Star

Figure 7.9e depicts ϵ-NFA construction for applying unary operator Kleene star for *zero* or more occurrences of machine M_1. Extra nodes are added before and after the initial and final nodes of the machine. An ϵ-transition from the initial node to the final node is added when M_1 does not occur in the input string. An ϵ-transition from the final node of M_1 to the initial node of M_1 allows for the necessary cycling in the matching process. The implementation is shown in Listing 7.6.

Kleene Plus

In terms of semantics, operator Kleene plus is similar to operator Kleene star. The difference is that machine M_1 is executed *one* or more times. The resulting machine, as depicted in Figure 7.9f, looks more like the machine for optional composition with the ϵ-transition reversed. The implementation is shown in Listing 7.7.

```
public class KleenePlus extends UnaryOperator
{
  public Object clone() {
    PatternExpression copy_lhs = lhs.clone();
    KleenePlus copy_self = (KleenePlus)super.clone();
    copy_self.addLHS(copy_lhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    NFAMachine m1 = lhs.createNFA();
    NFAPatternNode m1_initial = m1.getInitialNode();
    NFAPatternNode m1_final = m1.getAcceptingNode();
    Symbol epsilon = (Symbol)Epsilon.Instance();
    m1_final.addTransition(new Transition(epsilon, m1_initial));
    return new NFAMachine(m1_initial, m1_final);
  }
}
```

Listing 7.7: Implementation of Kleene plus operator

Repetition

A fixed number of repetitions is defined as a chain of sequential composition operators. The implementation is shown in Listing 7.8. It shows a recursive definition of `createNFA`, which keeps adding a sequence in front of the repetition until there are no repetitions left.

Counters

Section 5.4.4 introduced counters into the pattern language. The added expressiveness, however, made the language non-regular. To be able to use standard algorithms for NFA to DFA conversion, we implement a counter as an action executed when entering a node, instead of not on a transition. To do this, a counter expression is considered a Kleene star operation that is annotated with actions. Figure 7.9e depicts an ϵ-NFA for a Kleene star operation on machine M_1.

Each counter has a unique counter ID. The machine keeps track of all counters by using a dictionary, mapping counter names to a list of counter IDs. The initial node of the ϵ-NFA resets the counter. The final node of machine M_1 increases the counter. The final node of the new ϵ-NFA contains a condition to evaluate if the counters have the same value and if they are above the minimum value. The minimum value is used for counter conditions (such as ≥ 0 and > 0). The implementation is shown in Listing 7.9.

Whenever two machines are composed during NFA construction, the condition attached to an accepting state is copied to the accepting state of the new machine. This is done by method `mergeFinalActions` and should be present in the code for all previous operators too (lines 33–35).

```
public class Repetition extends UnaryOperator
{
  private int count;

  public void setCount(int c) {
    count = c;
  }

  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.clone();
    Repetition copy_self = (Repetition)super.clone();
    copy_self.setCount(count);
    copy_self.addLHS(copy_lhs);
    return copy_self;
  }

  public NFAMachine createNFA() {
    if (count > 1) {
      PatternExpression copy_lhs = (PatternExpression)lhs.clone();
      Sequence one = new Sequence();
      one.addLHS(copy_lhs);
      Repetition loop = new Repetition(count - 1);
      one.addRHS(loop);
      return one.createNFA();
    } else if (count == 1) {
      return lhs.createNFA();
    } else {
      NFAPatternNode empty = new NFAPatternNode();
      return new NFAMachine(empty, empty);
    }
  }
}
```

Listing 7.8: Implementation of repetition operator

```
public class Counter extends UnaryOperator {
  private String countername;

  public Object clone() {
    PatternExpression copy_lhs = (PatternExpression)lhs.clone();
    Counter copy_self = (Counter)super.clone();
    copy_self.addLHS(copy_lhs);
    return copy_self;
  }

  public void setName(String n) {
    name = n;
  }
  public void setMinimum(int i) {
    minimum = i;
  }

  public NFAMachine createNFA() {
    PatternAction a_reset, a_inc, a_c;
    NFAMachine m1 = lhs.createNFA();
    // code to create machine with Kleene star

    if(!m1.hasCounter(name)) {
      a_c = new PatternConditionCompareCounters(name, minimum);
      finalNode.addCondition(c);
    }
    string id = m1.addCounter(name);
    a_reset = new PatternActionResetCounter(id);
    a_inc = new PatternActionIncreaseCounter(id);
    initialNode.addAction(a_reset);
    m1_final.addAction(a_inc);

    NFAMachine m = new NFAMachine(initialNode, finalNode);
    m.moveFinalActions(m1);
    return m;
  }
}
```

Listing 7.9: Implementation of counter operator

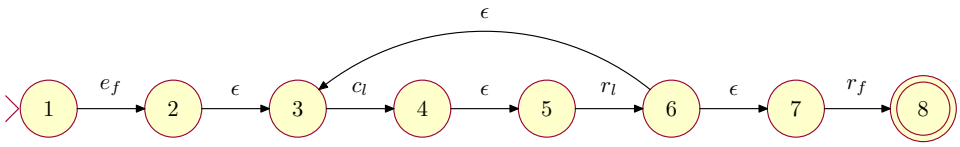


Figure 7.10: ϵ-NFA of change pattern from inventory example

7.4.2 Example NFA of Inventory Pattern

We will illustrate the NFA construction process by building the automaton for our change pattern from the inventory example. The abstract syntax tree for our pattern is previously shown in Figure 7.2b. By calling method `createNFA` on the root node of this parse tree, an NFA is build by a depth first traversal. The resulting ϵ-NFA is shown in Figure 7.10. The transition between states 1 and 2 represents the execution of a function on class `Inventory`. Transitions between state 3 and 6 represent the first, and consecutive, pairs of call and return to a state changing method on variable `stock`. Finally, the transition between state 7 and 8 represents a return from the method execution on class `Inventory`. State 8 is the final state.

The next section explains how nondeterminism is removed by converting this automaton to a DFA.

7.4.3 Conversion to Deterministic Finite Automaton

Acceptance in a nondeterministic automaton is determined by processing an input string and reaching an accepting state. In an ϵ-NFA there may be several paths that represent the processing of an input string, while a DFA only contains one such path. To remove nondeterminism, the DFA simulates the simultaneous exploration of all possible paths in the ϵ-NFA. The nodes in a DFA correspond to sets of nodes in the ϵ-NFA.

We illustrate the conversion by using the automaton of our change pattern, as depicted in Figure 7.10. To explain the given algorithms, the automaton is represented as a state transition table (STT) in Table 7.1.

Table 7.1: Transition table for transition function δ of ϵ-NFA from Figure 7.10

	Σ_ϵ				
Q	ϵ	e_f	c_l	r_l	r_f
1	$\emptyset$	$\{2\}$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\{4\}$	$\emptyset$	$\emptyset$
4	$\{5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\emptyset$
6	$\{3, 7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\emptyset$	$\{8\}$	$\emptyset$
8	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Formally, an ϵ-NFA $\mathcal{M}$ is represented by a five tuple: $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ_ϵ the input alphabet (our symbols plus ϵ), initial state q_0 and a set of accepting states F. An STT, like the one given in Table 7.1, shows transition function δ. For an NFA, it is a function from $Q \times \Sigma_\epsilon$ to $\mathcal{P}(Q)$, the power set of Q, where $\delta(q_i, a)$ returns a set of nodes reachable from state q_i on symbol a.

From the transition function δ for an NFA, the corresponding transition function t for the DFA can be constructed. The definition of that function $t(q_i, a)$ can be broken into three parts [Sudkamp, 1997, page 175]. First, the set of states that can be reached from q_i without processing a symbol, called the *epsilon-closure* of state q_i. This is followed by processing an a from all states in that set. Finally, the ϵ-closure of that set of states yields the set $t(q_i, a)$. From this transition function, a DFA is build. The results of the first and third part are shown in Table 7.2. Before presenting the deterministic automaton, we first give the algorithms for constructing the ϵ-closure and transition function t.

Epsilon-Closure

The ϵ-closure of a state q_i is defined recursively by [Sudkamp, 1997, page 176]

i) Basis step: $q_i \in \epsilon\text{-closure}(q_i)$
ii) Recursive step: Let $q_j \in \epsilon\text{-closure}(q_i)$. If $q_k \in \delta(q_j, \epsilon)$, then $q_k \in \epsilon\text{-closure}(q_i)$.
iii) Closure step: q_j is in $\epsilon\text{-closure}(q_i)$ only if it can be obtained from q_i by a finite number of applications of the recursive step.

Table 7.2a shows the ϵ-closure for every state in Table 7.1.

For our implementation the recursive definition is turned into a loop, as shown in Listing 7.10. A queue is used to keep track of the list of nodes to be checked for ϵ-transitions. We start of by adding the first node to both the queue and the set of found closure nodes. In a loop each element is removed from the queue and checked for an ϵ-transition. If found, that node is added to the queue and to the set of found closures. When the queue is empty, all ϵ-transitions are found.

Table 7.2: Transition table for the ϵ-closure and input transition function corresponding to Table 7.1

(a) ϵ-closure

Q	**ϵ-closure**
1	$\{1\}$
2	$\{2,3\}$
3	$\{3\}$
4	$\{4,5\}$
5	$\{5\}$
6	$\{6,3,7\}$
7	$\{7\}$
8	$\{8\}$

(b) Input transition function t

	Σ			
Q	e_f	c_l	r_l	r_f
1	$\{2,3\}$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{4,5\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\{4,5\}$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{6,3,7\}$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\{6,3,7\}$	$\emptyset$
6	$\emptyset$	$\{4,5\}$	$\emptyset$	$\{8\}$
7	$\emptyset$	$\emptyset$	$\emptyset$	$\{8\}$
8	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

```
HashSet epsilonClosure(PatternNode initial_node) {
  LinkedList queue = new LinkedList();
  HashSet closure = new HashSet();
  PatternNode node, next_node;
  Transition transition;
  Iterator transiter;
  Symbol epsilon = (Symbol)Epsilon.Instance();

  closure.add(initial_node);
  queue.add(initial_node);

  while(!queue.isEmpty()) {
    node = (PatternNode)queue.removeFirst();
    transiter = node.transitionIterator();

    while(transiter.hasNext()) {
      transition = (Transition)transiter.next();

      if(transition.getSymbol().equals(epsilon) {
        next_node = transition.getNextNode();

        if(!closure.contains(next_node)) {
          closure.add(next_node);
          queue.addFirst(next_node);
        }
      }
    }
  }
  return closure;
}
```

Listing 7.10: Implementation for ϵ-closure of a node

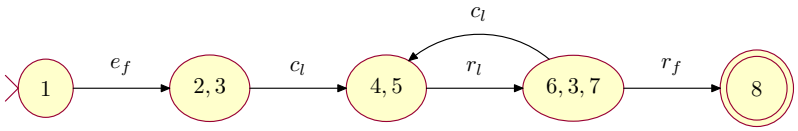


Figure 7.11: Deterministic automaton of change pattern created from ϵ-NFA

Subset Construction

An *input transition function* t uses an algorithm called *subset construction* to define which states from the ϵ-NFA are merged into an equivalent state in the DFA. For an ϵ-NFA $\mathcal{M}$, t is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ and defined as [Sudkamp, 1997, page 176].

$$t(q_i, a) = \bigcup_{q_j \in \epsilon\text{-closure}(q_i)} \epsilon\text{-closure}(\delta(q_j, a)) \tag{7.1}$$

where δ is the transition function of $\mathcal{M}$. For an NFA without ϵ-transitions, the input transition function t is identical to the transition function δ of the automaton. In its worst case, the algorithm creates 2^n states for an NFA with n states. Table 7.2b shows the input transition function for every state in Table 7.1. The resulting DFA is shown in Figure 7.11.

How the algorithm is implemented is shown in Listing 7.11. Method `createDFA` receives an NFA machine as argument and will return a DFA machine. It starts by calculating the ϵ-closure of the initial state of the ϵ-NFA. The result is put in a queue, which represents the list of grouped NFA nodes to consider next. When the list is empty, the automaton is deterministic. When a group of NFA nodes is taken from the queue, all transitions on each containing node are iterated one by one. On each transition of a symbol, the ϵ-closure is applied. If the resulting group of NFA nodes was not created by a previous iteration, it is added to the queue and a new equivalent DFA node is created. The DFA node is build by merging grouped NFA nodes and adding a transition to the previous node in the constructed DFA.

Subset construction maps a set of NFA nodes to a new DFA node. The new DFA node should have the same behavior as the union of the NFA nodes. All properties are be merged. It does *not* merge transitions, however. The DFA node is accepting when one of the NFA nodes is accepting. All actions and conditions are copied to the new DFA node. The implementation of method `mergeNodes` is given in Listing 7.12.

7.4.4 Minimization of Automaton

The subset construction algorithm, explained in Section 7.4.3, has an upper bound of creating 2^n states for an NFA with n states. In general this worst case rarely occurs. According to Leslie [1995], exponential blowup can be expected for non-deterministic automata that have a *deterministic density* of around two. Deterministic density dd is defined as the number of transitions p divided by the number of states n multiplied by the number of symbols m.

$$dd = \frac{p}{nm} \tag{7.2}$$

```
DFAMachine createDFA() {
  LinkedList queue = new LinkedList();
  HashSet nfagroupnode, new_nfagroupnode, all_nfagroupnodes;
  HashMap nfagroupnode_map = new HashMap();
  Iterator nodeiter, transiter;
  PatternNode nfanode, next_nfanode;
  DFAPatternNode dfanode, new_dfanode, initial_dfanode;
  Transition transition;
  Symbol epsilon = (Symbol)Epsilon.Instance();
  DFAMachine machine;

  new_nfagroupnode = epsilonClosure(getInitialNode());
  all_nfagroupnodes.add(new_nfagroupnode);
  queue.add(new_nfagroupnode);
  initial_dfanode = new DFAPatternNode();
  nfagroupnode_map.put(new_nfagroupnode, initial_dfanode);
  mergeNodes(initial_dfanode, new_nfagroupnode);
  machine = new DFAMachine(initial_dfanode);

  while(!queue.isEmpty()) {
    nfagroupnode = (HashSet)queue.removeFirst();
    nodeiter = nfagroupnode.iterator();

    while(nodeiter.hasNext()) {
      nfanode = (PatternNode)nodeiter.next();
      transiter = nfanode.transitionIterator();

      while(transiter.hasNext()) {
        transition = (Transition)transiter.next();

        if(!(transition.getSymbol().equals(epsilon))) {
          next_nfanode = transition.getNextNode();
          new_nfagroupnode = epsilonClosure(next_nfanode);

          if(!all_nfagroupnodes.contains(new_nfagroupnode)) {
            all_nfagroupnodes.add(new_nfagroupnode);
            queue.add(new_nfagroupnode);
            new_dfanode = new DFAPatternNode();
            nfagroupnode_map.put(new_nfagroupnode, new_dfanode);
            mergeNodes(new_dfanode, new_nfagroupnode);
          }
          dfanode = (DFAPatternNode)nfagroupnode_map. get(
              nfagroupnode);
          new_dfanode = (DFAPatternNode)nfagroupnpde_map. get(
              new_nfagroupnode);
          dfanode.addTransition(new Transition(new_dfanode,
              transition.getSymbol()));
        }
      }
    }
  }
  return machine;
}
```

Listing 7.11: Implementation for ϵ-NFA to DFA conversion

```
void mergeNodes(PatternNode target, HashSet nodes) {
  Iterator nodeiter, transiter, actioniter, conditer;
  PatternNode node;

  nodeiter = nodes.iterator();
  while(nodeiter.hasNext()) {
    node = (PatternNode)nodeiter.next();

    if(!target.equals(node)) {
      if(node.isAccepting()) {
        target.setAccepting(true);
      }
      actioniter = node.actionIterator();
      while(actioniter.hasNext()) {
        target.addAction((PatternAction)actioniter.next());
      }
      conditer = node.conditionIterator();
      while(conditer.hasNext()) {
        target.addCondition((PatternCondition)conditer.next());
      }
    }
  }
}
```

Listing 7.12: Implementation for merging a set of nodes in a target node

The algorithm is meant for an ϵ-free NFA, where the number of transitions do not contain duplicates and do not lead to a fail state. The deterministic density represents the ratio of the number of transitions in a given NFA to the number of transitions of a fully connected DFA having the same number of states and transitions.

After an automaton has been made deterministic using subset construction, it often contains equivalent states which can be merged to form a smaller DFA, while accepting the same language.

Equivalent States

Consider the automaton shown in Figure 7.12. It is generated from the following pattern expression.

$$e_f c_l c_r (c_l c_r)^* r_f \tag{7.3}$$

It recognizes the same language as our previous examples of state changing function of inventory. This pattern makes the Kleene plus operator explicit by chaining a sequence operator with a Kleene star operator. The DFA is equivalent to the one depicted in Figure 7.11. By merging equivalent states, we can minimize a DFA.

Two states that are equivalent are called *indistinguishable*. Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a DFA over states Q, symbols Σ, transition function δ, initial state $q_0 \in Q$ and final states $F \subseteq Q$. States q_i and q_j are *distinguishable* if there is a string u such that $\hat{\delta}(q_i, u) \in F$ and $\hat{\delta}(q_j, u) \notin F$, or vice versa [Sudkamp, 1997, page 183]. Function $\hat{\delta}$ simulates the repeated application of transition function δ

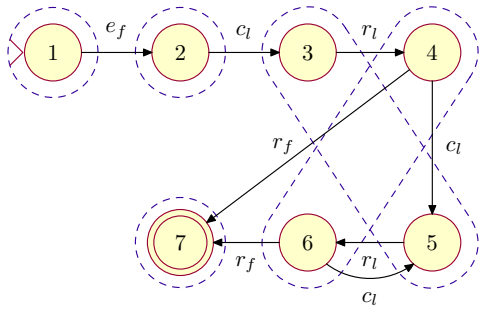


Figure 7.12: Non-minimal DFA of pattern with equivalent states marked

using an input symbol from string u.

To merge equivalent states, we need to recognize equivalent states. We can define an *Myhill-Nerode* equivalence relation $\equiv_{\mathcal{M}}$ for language $\mathcal{L}(\mathcal{M})$ as follows [Sudkamp, 1997, pages 217–222]

$$u \equiv_{\mathcal{M}} v \iff \hat{\delta}(q_0, u) = \hat{\delta}(q_0, v) \tag{7.4}$$

for all strings u and v in Σ^* (all strings over Σ). When defining an equivalence relation over a set, one can partition that set into a collection of subsets, called *equivalence classes*. Any minimal DFA $\mathcal{M}$ has precisely as many states as the number of equivalence classes in Σ^* with respect to $\equiv_{\mathcal{M}}$.

This brings the problem of minimizing a DFA to finding the coarsest partition of a set of states. This is called *partition refinement*. The initial partitioning is into two sets, called blocks, the non-accepting and accepting states. Each block is refined (by splitting) in such a way that for any two states q_i and q_j that are in the same block, $\delta(q_i, a)$ and $\delta(q_j, a)$ go to the same block for each symbol a.

We will illustrate the algorithm by applying it to the DFA depicted in Figure 7.12. After partitioning the states into two blocks, the non-accepting and accepting states, we examine each state for equivalence. Each step of the partition refinement process is shown in Table 7.3. In the first block, state 1 has a transition on e_f to the first block, the other states (2 up to 6) do not, thus state 1 forms a new block on its own. The same applies to state 2. In the third iteration, state 3 and 5 both have a transition on r_l to the block they belong to, state 4 and 6, however, do not. In the fourth iteration every block contains equivalent states.

Table 7.3: Minimizing DFA of Figure 7.12 using partition refinement

Iteration	**Partition**
1	$\{\{1,2,3,4,5,6\}, \{7\}\}$
2	$\{\{2,3,4,5,6\}, \{7\}, \{1\}\}$
3	$\{\{3,4,5,6\}, \{7\}, \{1\}, \{2\}\}$
4	$\{\{4,6\}, \{7\}, \{1\}, \{2\}, \{3,5\}\}$

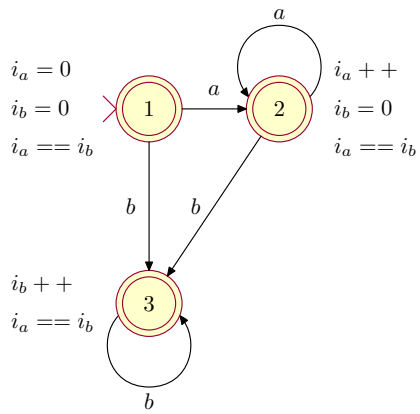


Figure 7.13: Automaton of pattern $a^i b^i$ with actions and conditions

Minimizing Automata with Actions

Counters were implemented as actions to nodes. When the automaton was made deterministic, the semantics of the automaton for counters did not change. The subset construction algorithm removed all ϵ-transitions by repeatedly calculating the ϵ-closure. The minimization process however, does not preserve the semantics of counters. This is demonstrated in Figure 7.13, which depicts an automaton that is extended with counters corresponding to pattern expression $a^i b^i$. It is constructed from the expression $a^* b^*$ using subset construction on the ϵ-NFA. State 1 resets all counters, states 2 and 3 have a counter that is incremented on every visit. Each state is accepting if the condition $i_a == i_b$ is satisfied.

Using the minimization algorithm would merge state 1 and state 2, they are equivalent in terms of possible transitions. The actions, however, would conflict. A node can either increase a counter or reset it, not both.

The equivalence relation is adjusted to compare each node on both actions and transitions.

Minimizing a DFA

The implementation for minimizing a DFA is given in Listing 7.13. First we partition the DFA into non-accepting and accepting nodes. We can implement those methods by searching from the initial node, or we can modify line 19 and line 40 of Listing 7.11 to register them. Method `split` is called in a loop to refine each block in the partition until no further refinements can be made. In that case, all equivalence classes are found. If a block was refined, the partition has changed and the iterator needs to be reset. When the partition can not be refined any further, the nodes of every block are merged by calling method `createMinimalDFA`.

```
Machine minimize() {
  LinkedList partition = new LinkedList();
  HashSet block;
  Iterator partiter;
  boolean refined = true;

  block = new HashSet(getNonAcceptingNodes());
  partition.add(block);
  block = new HashSet(getAcceptingNodes());
  partition.add(block);

  while(refined) {
    refined = false;
    partiter = partition.iterator();

    while(partiter.hasNext()) {
      block = (HashSet)partiter.next();
      refined = split(block, partition);

      if(refined) {
        break;
      }
    }
  }
  return createMinimalDFA(partition);
}
```

Listing 7.13: Implementation for making a DFA minimal

Block Refinement

Block refinement is done by method `split`. The implementation is given in Listing 7.14. The first argument is the block to check for equivalence. A block contains a set of nodes. We compare the first node of a block with the other nodes from that block. Method `createLookupTable` creates a lookup table of all transitions of the first node and maps them to a block from the partition containing the target node. By looping the other nodes of the block, we compare whether the target of each transition belongs to the same block as for the transition on the same symbol of the first node. If both nodes point to the same block on the same symbol, the nodes are equal. If the actions of both nodes are the same, the node is added to the `equal` set. If not, the node is added to the `inequal` set. Before looping all transitions, we can check if the degree of the first node equals the degree of the other node. The degree of a node is the number of outgoing transitions. If it differs, both nodes are different and we do not need to iterate over every transition. The same applies for the number of actions. If the set of inequal nodes is empty, the block was already a minimal equivalence class. If it was not, the block is removed from the partition and the refined set is added along with the set of inequal nodes.

The implementation for creating a lookup table is given in Listing 7.15. It iterates all transitions of the given node and stores each transition as a map of symbols to the block containing the target node of that transition.

Merging Nodes Inside Blocks

When all equivalent classes of a DFA have been found, each block inside a partition corresponds to a node of the minimal DFA. Listing 7.16 shows the implementation for merging nodes inside a block. The initial node of every block will become a node in the minimal DFA. By means of a lookup table, every node in a block will point to the initial node of that block. Every initial node is added to a queue. By calling method `mergeNodes`, final states and conditions are merged. The implementation of `mergeNodes` for a DFA does not merge actions. When the lookup table is filled, every transition of every node in the queue is examined. From the information found in the lookup table, every next node of a transition is corrected to point to a node found in the minimal DFA.

```
boolean split(HashSet block, LinkedList partition) {
  HashSet equal = new HashSet();
  HashSet inequal = new HashSet();
  HashMap lookup;
  Iterator nodeiter, transiter, actioniter
  PatternNode splitnode, node, nextnode;
  Transition transition;
  Symbol symbol;
  int degree, bool ok;

  nodeiter = block.iterator();
  if(nodeiter.hasNext()) {
    splitnode = (PatternNode)nodeiter.next();
    equal.add(splitnode);
    lookup = createLookupTable(splitnode, partition);
    degree = splitnode.getDegree();
    actions = splitnode.getNumberOfActions();
    while(nodeiter.hasNext()) {
      node = (PatternNode)nodeiter.next();
      if(degree == node.getDegree() || actions == node.
          getNumberOfActions()) {
        transiter = node.transitionIterator();

        while(transiter.hasNext()) {
          transition = (Transition)transiter.next();
          symbol = transition.getSymbol();
          nextnode = transition.getNextNode();
          pblock = (HashSet)lookup.get(symbol);
          if(pblock.contains(nextnode)) {
            ok = true;
            actioniter = splitnode.actionIterator();
            while(actioniter.hasNext()) {
              action = (PatternAction)actioniter.next();
              if(!node.contains(action)) ok = false;
            }
            if(ok) equal.add(node); else inequal.add(node);
          } else {
            inequal.add(node);
          }
        }
      } else {
        inequal.add(node);
      }
    }
  }
  if (inequal.isEmpty()) {
    return false;
  } else {
    partition.remove(block);
    partition.addFirst(inequal);
    partition.addLast(equal);
    return true;
  }
}
```

Listing 7.14: Implementation for splitting a block of a partition

```
HashMap createLookupTable(PatternNode node, LinkedList partition) {
  HashMap lookup = new HashMap();
  HashSet block;
  PatternNode nextnode;
  Symbol symbol;
  Iterator transiter, partiter;

  transiter = node.transitionIterator();

  while(transiter.hasNext()) {
    transition = (Transition)transiter.next();
    symbol = transition.getSymbol();
    nextnode = transition.getNextNode();
    partiter = partition.iterator();

    while(partiter.hasnext()) {
      block = (HashSet)partiter.next();

      if(block.contains(nextnode)) {
        lookup.put(symbol, block);
      }
    }
  }
  return lookup;
}
```

Listing 7.15: Implementation for creating a lookup table mapping transitions to blocks

```
Machine createMinimalDFA(LinkedList partition) {
  LinkedList queue = new LinkedList();
  HashMap lookup = new HashMap();
  HashSet block;
  Iterator partiter, nodeiter;
  PatternNode firstnode, node;

  partiter = partition.iterator();
  while(partiter.hasNext()) {
    block = (HashSet)partiter.next();
    nodeiter = block.iterator();
    firstnode = (PatternNode)nodeiter.next();
    mergeNodes(firstnode, block);
    queue.add(firstnode);

    while(nodeiter.hasNext()) {
      node = nodeiter.next();
      lookup.put(node, firstnode);
    }
  }

  while(!queue.isEmpty()) {
    firstnode = (PatternNode)queue.removeFirst();
    transiter = firstnode.transitionIterator();

    while(transiter.hasNext()) {
      transition = (Transition)transiter.next();
      node = lookup.get(transition.getNextNode());
      transition.setNextNode(node);
    }
  }

  firstnode = lookup.get(getInitialNode());
  return new Machine(firstnode);
}
```

Listing 7.16: Implementation for creating a DFA from a minimal partition

```
void total(Vector symbols) {
  LinkedList queue = new LinkedList();
  DFAPatternNode node, nextnode;
  Transition transition;
  Iterator symboliter;

  queue.add(getInitialNode());
  fail = new DFAPatternNode();
  fail.setFailing(true);

  while(!queue.isEmpty()) {
    node = (DFAPatternNode)queue.removeFirst();
    symboliter = symbols.iterator();

    while(symboliter.hasNext() {
      symbol = (Symbol)symboliter.next();
      transition = node.getTransition(symbol);

      if(symbol.equals(transition.getSymbol())) {
        nextnode = (DFAPatternNode)transition.getNextNode();

        if(!queue.contains(nextnode)) {
          queue.add(nextnode);
        }
      } else {
        node.addTransition(new Transition(fail, symbol));
      }
    }
  }
}
```

Listing 7.17: Implementation for making a DFA total

7.4.5 Total Automaton

A total function is a function defined over all possible inputs. Symbols that are declared in the event module, but are not used in transitions of the automaton, cause the automaton to fail matching a pattern.

Listing 7.17 shows the implementation for making the transition function of a DFA total. A queue keeps track of all states we need to visit. When the queue is empty, the transition function is total. The set of declared symbols is passed as argument. For every node in the automaton, each symbol is looked up in the transitions for that node. If a transition for a declared symbol is not present, it is added. If a transition for a symbol is present, the node is added to the queue.

7.4.6 Discussion

This section explained how to build a minimal DFA from a pattern expression. There are, however, many more algorithms to accomplish the same task each with different behavior with respect to time and memory.

Leslie [1995] and van Noord [2000] studied various subset construction algorithms for complexity. They defined density relations over the number of states and the number of ϵ-moves, respectively, and related that to various algorithms to characterize expected efficiency.

The minimization algorithm presented in this section is based on a partition refinement algorithm for bisimulation equivalence by Kanellakis and Smolka as discussed by Cleaveland and Sokolsky [2001, pages 398–400]. The algorithm is modified to first iterate over all nodes of a block, then over all symbols. The other difference is using a lookup table instead of sorting the list of transitions. Our automata have actions attached to a node, they needed to be included for testing equivalence. The algorithm by Kanellakis and Smolka expect a total automaton, where the degree of each state is the same. By postponing the step of creating a total automaton, we can use the degree to distinguish between states. The time complexity is $\mathcal{O}(nm)$ for a DFA with n states and m symbols. The extra loop to compare actions makes the complexity $\mathcal{O}(nma)$ for a actions. A faster algorithm was developed by Paige and Tarjan with time complexity $\mathcal{O}(m \log_2 n)$, also discussed by Cleaveland and Sokolsky [2001]. The algorithm by Paige and Tarjan, however, needs a more complex data structure. It traverses the edges in opposite direction and build a binary tree of blocks.

Counters were added as action to nodes. An alternative approach would be to add them as action to transitions. This would, however, complicate the ϵ-closure in merging transitions. A benefit would be the addition of guards on transitions. Checking for conditions in the final state only, has the downside of not terminating a pattern when halfway through a condition is violated. If a counter in an expression is made optional, the condition in the final state is still checked, causing the automaton to fail if the condition does not hold.

The algorithms presented in this section are meant as proof of concept, not as the most optimal implementation. Each algorithm will result in the same minimal DFA, as stated by the Myhill-Nerode theorem. By using incremental compilation, as proposed by Spenkelink [2006], a previously minimized DFA can be kept between successive runs of the Compose$\star$ compile-time. Given that the pattern expression was not modified.

This concludes the compile-time part for storing event modules in the repository.

7.5 Instrumentation

For monitoring events at run-time, the base code needs to be *instrumented* to signal events when they occur. Instrumentation is done by weaving hooks in the byte code. The decision to weave inside the byte code is motivated in Section 5.1.

Weave points are all potential places in the program where hooks can be woven in [Dürr et al., 2005]. An *event join point* is a weave point in the program which is instrumented for signalling events. The join point model contains all possible types of join points, for events they are discussed in Section 4.3.

The places in the code that need to be instrumented can be identified by analyzing the event modules stored in the repository. Each symbol corresponds to an event type. Because of local monitoring, only those classes that are specified in superimposition are considered for instrumentation.

The compile-time created an abstract syntax tree (AST) of the base program (including compiled advices). The AST consists of intermediate language statements. A naive weaver would instrument all possible occurrences of each event type in the AST of the superimposed classes and imposed internals. By analyzing the properties of each symbol, the set of weave points can be refined. Consider for example the symbols `Cl` and `Rl`, that are used in our inventory example of Section 6.2. They have a target property of `inner.stock`. This means we only need to instrument the calls to system class `ArrayList` that operate on variable `stock`. And only on those methods that have been selected by superimposition (lines 5–8 of Listing 6.4).

Each symbol, however, appears in a certain context in the pattern. This ordering can be used to further refine the set of weave points. Through control flow analysis on the AST, each path leading up to a certain symbol within a class can be analyzed. Only those paths that conform to the pattern have to be instrumented. At run-time, however, each pattern remembers its state in between calls to the object. Thus we have to consider all paths that are reachable from outside the object too.

The current weaver of Compose⋆, as described in the M. Sc. thesis of Boschman [2006], contains support for weaving hooks on various types of method invocation, object instantiation, application start and field access. Other join points, like selection and iteration for example, are not supported by the weaver. Field access is currently not used by Compose⋆. The weaver weaves hooks based on a weave specification. This specification contains the methods and object instantiations that have to be instrumented. A hook is woven in place of the call, and for object instantiation, after the instantiation. The hook itself is a static call to the message handling facility of Compose⋆. Until the weaver and weave specification generation parts are modified for the other join points, we can only reason about patterns describing a trace of messages.

For method calls that return a value, the signature of a hook for methods that return a value is given below.

```
1 static Object handleReturnMethodCall(Object caller, Object target,
    String selector, Object[] args)
```

The weaver passes a reference to the caller and target object, along with the

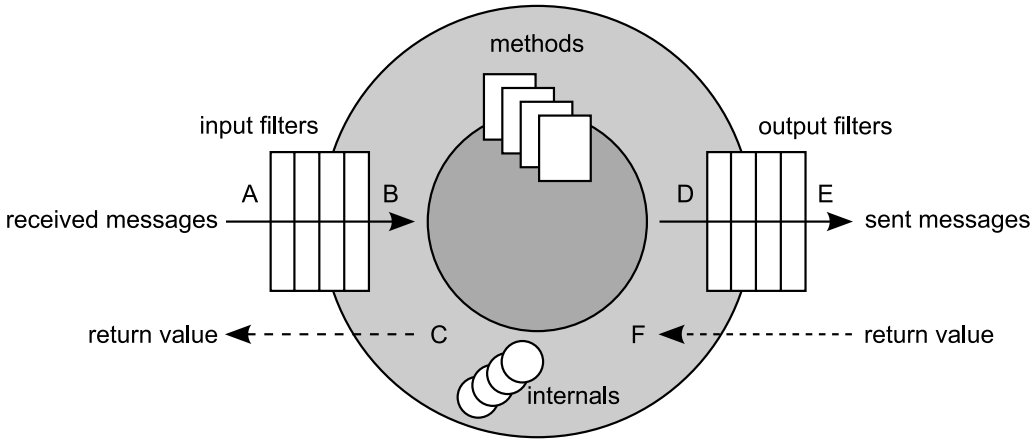


Figure 7.14: Local monitoring of events with respect to message filtering

name of the invoked method and an array of its arguments. From the arguments passed by a hook, the message handling facility creates a message object. All filter processing is done on this message. Depending on the outcome of filter processing, the (possibly altered) message is invoked using reflection mechanisms provided by the virtual machine.

Unlike messages, events can not be invoked. They signal that some action (type of event) is about to happen, or has happened. Which implies that the weaver should insert hooks before or after that action. The weaver will create the corresponding event, set the properties and call static method `handleEvent` in the event handling facility. The hook will look up the event manager and pass the event. For the corresponding *after event*, the *before event* can be reused by moving the old values and setting the new ones. The *after event* for data assignment, for example, contains the old and new value.

For message events, the message handling facility can be reused. When the run-time creates a message object, a corresponding event object is created. This enables generation of events at different stages of filter processing without altering the weaver. The downside is, however, that every message is checked for presence of event modules, which can have a significant impact on performance.

7.5.1 Specifying Events for Message Handling

In Section 4.3.1 four message events are defined, namely before and after events for message call and execution. Before and after parameter evaluation events are not supported. The message handling facility (and the weaver) of Compose* do not provide those hooks. Parameter values can be used for matching on properties of messages, however.

When an object sends an outgoing message, its output filters may modify the message. The same applies for an incoming message and input filters. This implies that there are two types of before messages: before and after filtering. Each event is listed below and depicted in Figure 7.14, we will refer to them as events A to F.

A) Before call original message;
B) Before call filtered message;
C) After call filtered and original message;
D) Before execution original message;
E) Before execution filtered message;
F) After execution filtered and original message.

A developer needs to specify which of these event types he wants to use in a pattern. If unspecified, a default type could be used. Each pattern, however, will most likely require different types of message events. This makes it hard to choose a default. Each combination of message events differs substantially in semantics, as described by the following list.

Match after filter modules
Matching after both filter modules (event B and E) makes events adhere to the result of filter handling. There might be a mismatch, however, when filters get updated and not the corresponding event patterns. This can cause a pattern to never match;

Match before filter modules
Matching before filter modules (event A and D) assumes any rewritten message adheres to the same contract as the original message. Error filters, however, create false positives on the matching process, as the messages is never invoked.

Match both before and after filter modules
Matching on both before and after filter modules (events A, B, D, and E) will avoid the problem of the first two alternatives. The downside is, however, that two message events are initiated for one actual message. In the implicit model, this will cause the automaton to reach its fail state, unless we ignore the second event.

Match after incoming filters and before outgoing filters
Event B and D monitor message events on the inner object only.

Match before incoming filters and after outgoing filters
Event A and E monitor message events as they reach the 'outer' layer of an object.

This is related to the discussion in Section 5.3 on event filter models, where it is decided to signal separate events for filter handling. Because it depends entirely on the problem at hand, we choose to not set a default. A developer has to explicitly state which message event is used in a pattern by setting boolean property `filtered`.

To identify the places inside the run-time that need instrumentation, the message handling process is explained next.

7.5.2 Message Handling

When a hook is executed at run-time, it enters the message handling facility of Compose⋆. A message is a communication between a sender object and a target object. For each object an object manager is created which contains the filter modules imposed on that object.

The control flow for calling an instrumented method is depicted in Figure 7.15. Events C, D and F correspond to the events identified in the previous section, their instrumentation is explained in Section 7.5.3. The message handling facility creates a message object which represents the original call by setting the selector, arguments, sender (caller), and server (target) properties. Through a set of static calls, the repository is checked for the presence of filter modules on the sending object. If present, the object manager is retrieved through the global object manager and the message is delivered for processing the outgoing message.

When the outgoing filters are processed, the (possibly modified) message is processed by the input filters of the target. If the repository contains filter modules for the target, the message is delivered to the object manager of the target. If the target did not contain any filter modules, the message is invoked through reflection.

The control flow for delivery of incoming and outgoing messages is depicted in Figure 7.16. Instrumentation for events B, C, E, and F are explained in Section 7.5.3. Filter processing takes place in a separate thread initiated by the object manager. Communication between message delivery and filter processing happens through a message queue and a response buffer. The response buffer stores the return value of the message. Both are implemented as synchronized buffers

Before a message is delivered in the message queue, the and inner and target properties are set to the sender and target, respectively. The message is added to the message queue and a consumer is started in a separate thread through a call to method `notifyMessageConsumer`.

The return value is read from the response buffer through method `getResponse`, which blocks until a response becomes available. The response is either an object representing the actual return value from a message invocation, an object of type `ComposeStarAction`, or a message object. If the object is an action, it was the result of a filter evaluation that has not yet been executed. If the message to be delivered was an outgoing message, the return value can be of type message. In that case the message passed the set of filters without resulting in an action. This can only happen if the message will be passed on to `deliverIncomingMessage` by the message handling facility.

Method `notifyMessageConsumer` starts a new thread to process a message for filtering. Class `ObjectManager` implements interface `ChildRunnable`. When a new thread is started, the method `run` is automatically invoked. A sequence diagram for that method is depicted in Figure 7.17. Instrumentation of events B, C, E, and F is explained in Section 7.5.3.

When available, a message is consumed from the message queue. It is passed on to method `receiveMessage`, which will iterate over all filter modules superimposed on the object. Using the default filter policy, currently class

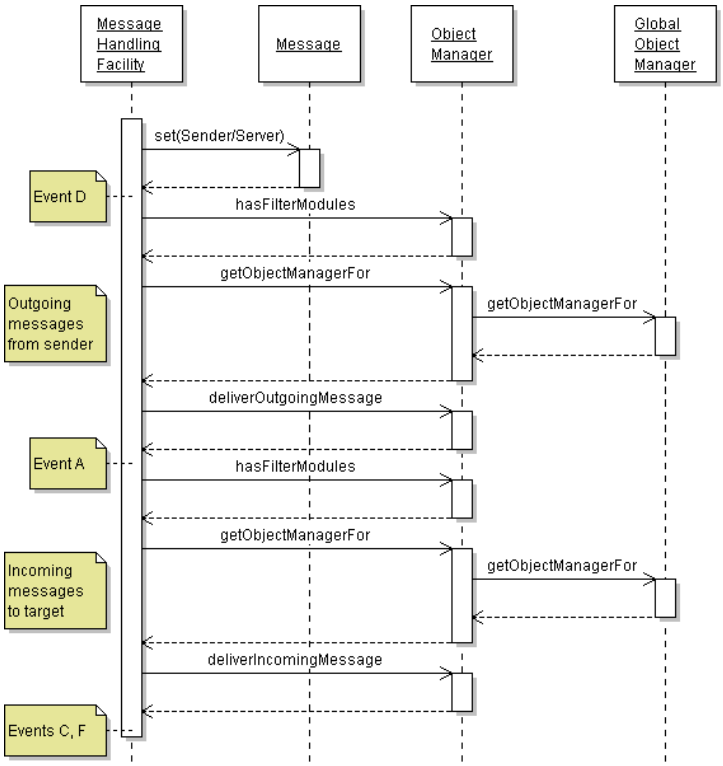


Figure 7.15: Sequence diagram for handling messages send from hooks at run-time

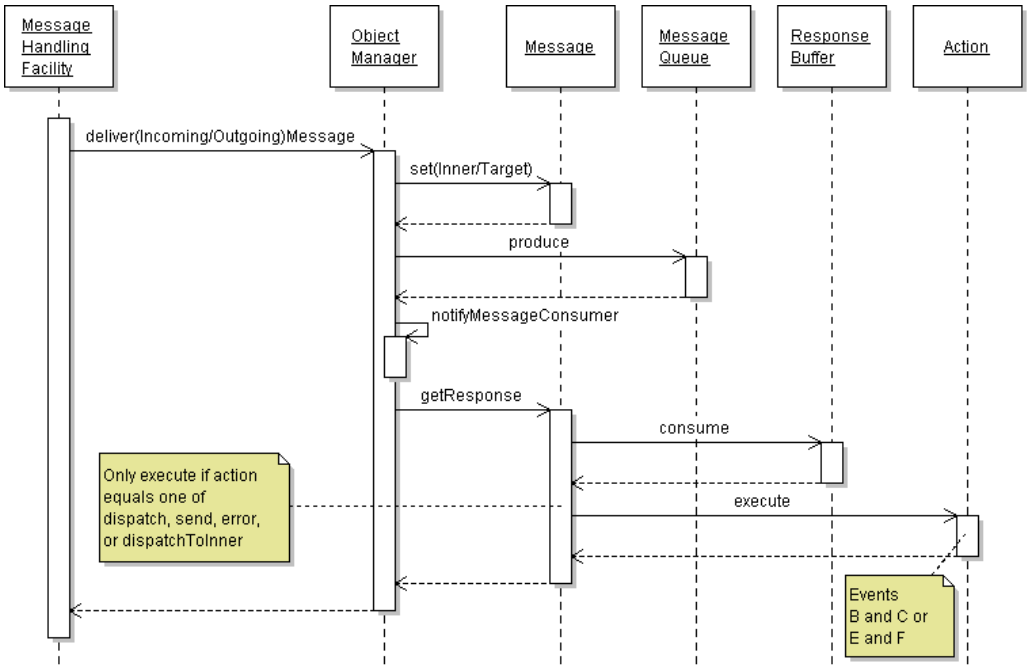


Figure 7.16: Sequence diagram for delivery of incoming and outgoing messages

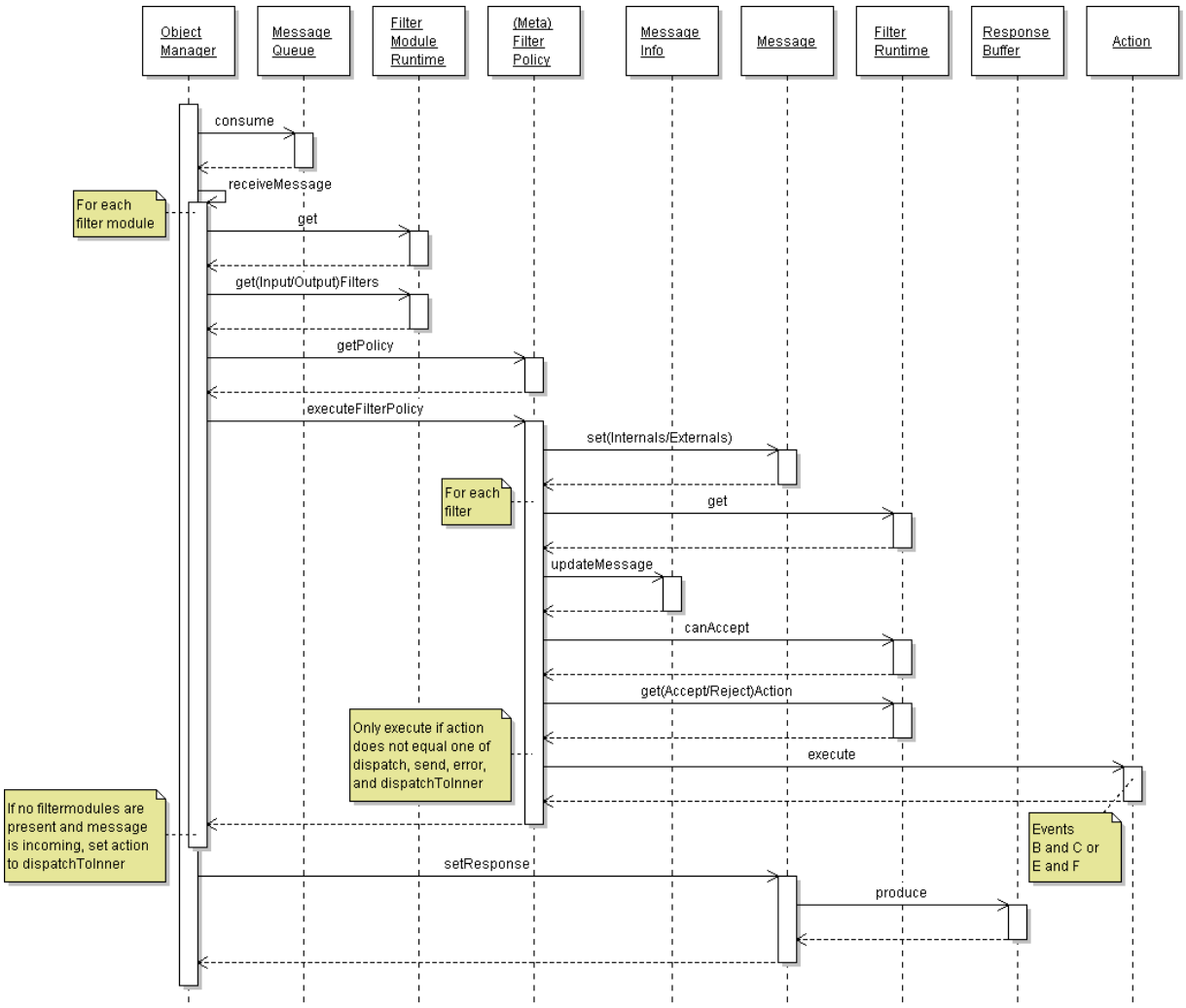


Figure 7.17: Sequence diagram for filtering messages

`MetaFilterPolicy`, each filter module is evaluated. For an incoming message, the set of input filters belonging to a filter module is passed to the policy, for an outgoing message, the set of output filters is passed.

Method `executeFilterPolicy` of class `MetaFilterPolicy` sets a reference to the internals and externals of a filter module in the current message. It will iterate over all filters in the module to evaluate them. Depending on the result, the filter accepts or rejects the message and sets an appropriate action as defined by the semantics of the filter (explained in Section 2.2). The action is executed if the action is *not* one of dispatch, dispatch to inner, send, and error, which means that currently meta and custom actions are executed inside the policy. In the next iteration the variables for storing the original message and modified message are updated. When filter processing is finished, the response is written into the response buffer, which is read by `getResponse` (discussed earlier for message delivery in Figure 7.16).

```
EventManager emSender, emTarget;
if(EventManager.hasEventModules(caller, datastore) {
  emSender = EventManager.getEventManagerFor(caller, datastore);
  emSender.queueEvent(new CallEvent(msg, Event.AFTER_ORIGINAL));
  emSender.handleEvent(new CallEvent(msg, Event.BEFORE_ORIGINAL));
}
// deliver incoming message
if(EventManager.hasEventModules(msg.getTarget(), datastore) {
  emTarget = EventManager.getEventManagerFor(msg.getTarget(), datastore);
  emTarget.queueEvent(new ExecEvent(msg, Event.AFTER_ORIGINAL));
  emTarget.handleEvent(new ExecEvent(msg, Event.BEFORE_ORIGINAL));
}
// deliver outgoing message
if(emTarget) emTarget.popAndHandleEvent();
if(emSender) emSender.popAndHandleEvent();
```

Listing 7.18: Changes for instrumenting a hook in the message handling facility

More information about the run-time and the message handling facility can be found in the M. Sc. theses of García [2003] and Staijen [2005].

7.5.3 Instrumenting Message Handling

To signal the message events discussed in the previous section, the message handling facility is instrumented.

Inside the message handling facility, a message is first delivered to the output filters that belong to the filter modules imposed on the sender. After that, the (possibly modified) message is delivered to the input filters that belong to filter modules imposed on the target. Before delivering the outgoing message, each hook is instrumented to raise a *before call original message* event. This is depicted in the sequence diagram of Figure 7.15 by event D. Before the outgoing message is filtered, an event of type A is raised, signaling a *before execution original message* event. When a response is given, the corresponding after events are signalled for both execution and call events. These after events must have their properties set to the original message. But after filtering, the original message may have been modified. By creating the after event at the same time as the before event, and storing it in a stack, the original properties are preserved. This reduces an after event to popping the stack as shown in Listing 7.18.

After delivering an incoming or outgoing message, an instance of class `MetaFilterPolicy` loops the imposed filter modules and executes each filter. The sequence diagrams are shown in Figure 7.16 and Figure 7.17 of section Section 7.5.2. If a filter results in a dispatch, dispatch to inner, send or error action, filter processing is finished. The action for dispatch and dispatch to inner can only occur inside incoming filters. The send action can only occur inside outgoing filters. The error action can be in both incoming and outgoing filters. Each action can be instrumented for raising before and after call and execution events. To signal these events, the `execute` method of each action is instrumented as described in the following list.

Dispatch An accept action for dispatch will invoke the method if the target is inner, or it will call a new message hook otherwise. If it invokes a method on inner, a *before execution filtered message* event is raised, and when the invoked method returns, the corresponding *after execution filtered message* event is raised. If the accept action calls a new message hook, a *before call filtered message* event is raised, and when it returns the corresponding *after call filtered message* is raised.

Dispatch to inner The dispatch to inner action acts the same as a normal dispatch action with an inner method as target.

Send An accept action for send marks the end for processing outgoing filters. The message continues to the incoming filters of the target. The `execute` method of this action, however, does not invoke a message, or call a new message hook. Because the next method call inside the message handling facility, after delivering an outgoing message, is delivering an incoming message. This makes it impossible to raise an after event. As a workaround, an *after call filtered message* event is added to event queue for later retrieval. An event for *before call filtered message* can be raised. When the hook inside the message handling facility calls method `popEvent`, it should check if the popped event is of type *after call filtered message*. If it is, another event should be popped.

Error An error filter is not a message event, it can be instrumented for raising an exception event, however.

Meta and custom Meta actions and custom actions differ from other actions. They are executed inside a different thread inside the filter loop of class `MetaFilterPolicy`. Events can not be raised unless class `EventManager` is modified to run inside its own thread and communicates through a buffer, analogous to class `ObjectManager`.

Discussion

Because of the complex architecture, it is difficult to adapt the run-time environment of Compose⋆ for event instrumentation. Instrumenting a send action needed a workaround to raise an *after call filtered message* event. For meta and custom actions, however, a more complex workaround is needed. These actions are executed inside a thread different from the base program. If a pattern matches because of events from the action, advice is also executed in the other thread. Extra synchronization and thread blocking is required in these cases.

Except for meta and custom filters, this section described how to instrument message handling from within the run-time environment. The next section describes how each event is processed by an event monitor.

7.6 Monitoring at Run-time

This section describes the event matching process inside the run-time environment of Compose⋆. First, the architecture is described, illustrated by class diagrams. Followed by a description of how each class interacts with each other using sequence diagrams.

7.6.1 Architecture for Monitoring Events

The architecture for handling events in the run-time is depicted in Figure 7.18.

When the base program instantiates an object with filters superimposed, it is coupled with an object manager and an event manager. An event manager contains a list of event modules that are imposed on the corresponding object, similar to an object manager and filter modules. The global object manager is modified to keep track of both object managers and event managers. They are stored inside a hash table using the object as key.

Events that are raised by instrumentation, are handled by calling method `handleEvent` on the event manager of an object. The event manager will delegate the event to its event modules for pattern matching. Methods `queueEvent` and `popAndHandleEvent` are used by instrumentation to store and retrieve *after events* for later processing. They are discussed in Section 7.5.3.

Each event module consists of a pattern (referenced through class `Machine`), a list of symbols, and a set of event filters. Event filters can refer to internals and externals based on certain conditions. They are also referenced by the event module, but not shown in the class diagram. They are the same entities as used by filter modules.

An event module is instantiated for each imposed object. Through superimposition, parameters can be passed at instantiation time. The value of a parameter can be a list of objects or a single object. They are represented by instances of class `SingleParameter` and `ListParameter` and should implemented according to the proposal for filter module parameters [Doornenbal, 2006].

An event module contains a reference to an automaton through class `Machine`. A machine contains a reference to the initial node of the automaton. Method `createPatternInstance` will instantiate a new pattern instance. A pattern instance keeps track of the 'state' of the machine and starts in the initial node. It contains a reference to the current node and maintains a dictionary of variables and counters.

Declared symbols are represented by instances of a subclass of abstract class `Symbol`. Their structure mimics that of events. A symbol contains properties. A property can be a value of some type of object, or a reference to a variable name. Class `Symbol` contains a method `compare` to match the symbol against an event. Next to the event to compare, that method expects a reference to a pattern instance to resolve variables.

Nodes in a machine consist of instances of class `PatternNode`. They contain a list of transitions to reach the next node, an optional set of actions and an optional set of conditions. An action is represented by an instance of a subclass

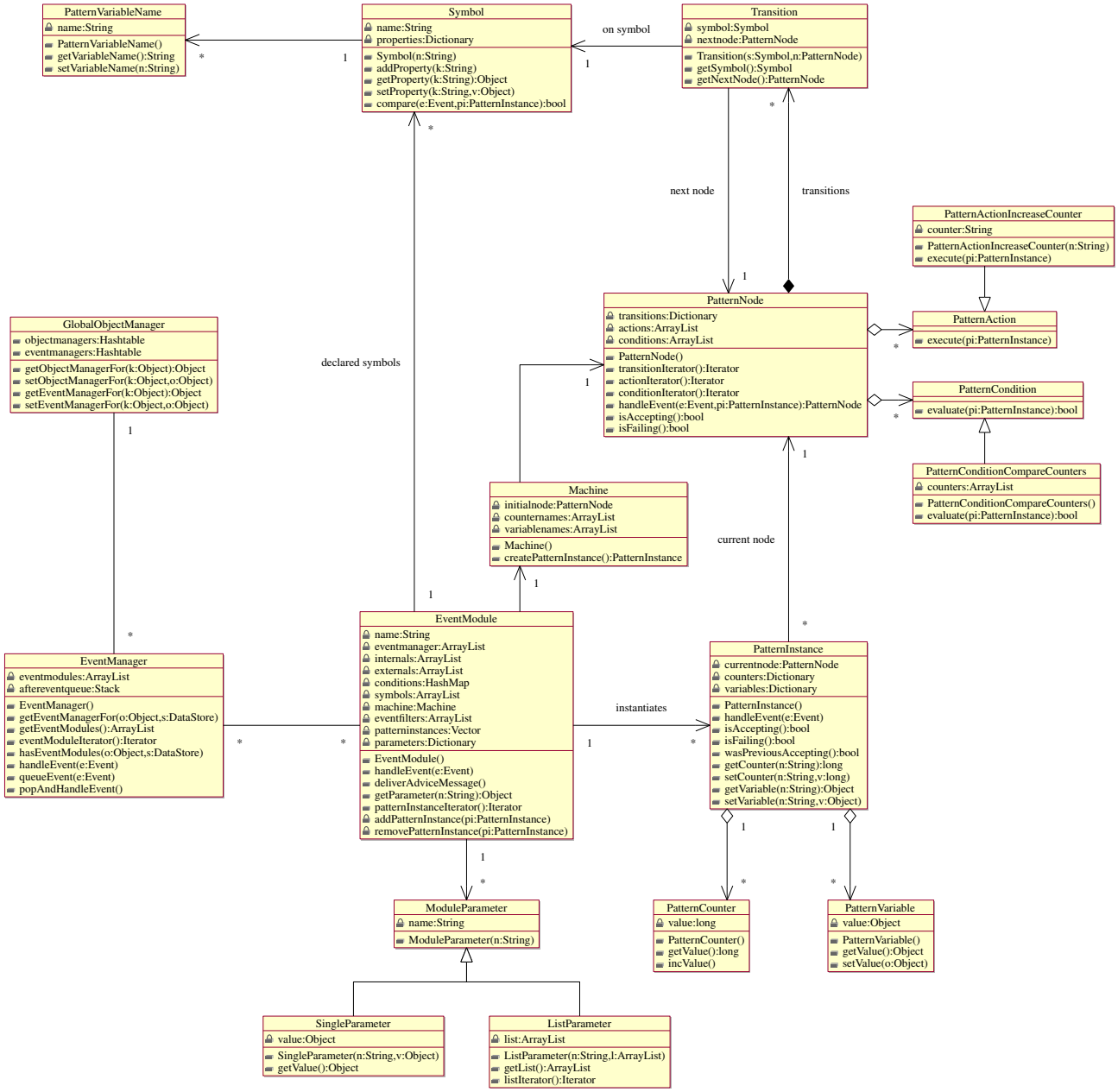


Figure 7.18: Class diagram of run-time architecture for event processing

of `PatternAction`. Currently, the only supported action is increasing the value of a counter. An action is executed when a pattern instance does a transition to a node containing an action. If a node is an accepting node, it can have a set of conditions. Conditions are represented by instances of a subclass of `PatternCondition`. Currently, the only supported condition is comparing counters for the same value.

If a machine reaches an accepting node and the condition evaluates to true, the event module will invoke some piece of advice as determined by event filters. Method `deliverAdviceMessage` of class `EventModule` is responsible delivering that message. Event filter evaluation is similar to message filter evaluation, with a few differences. The original message, which is used as input for the first event filter, is empty. It does not have a target and selector. Those values are to be filled in during evaluation. Event filters do not contain a matching part, only conditions and substitutions. A filter policy can be made to allow for different strategies of filter execution, but in practice, that feature is not actively used in Compose⋆. Every filter will have an accept and reject action similar to message filters. When the filter process is finished, the message is invoked.

7.6.2 Event Handling

The interaction between different classes used in the run-time for monitoring events are described using sequence diagrams.

Inside Event Manager

A sequence diagram for method `handleEvent` of class `EventManager` is depicted in Figure 7.19.

When an event is received, it is forwarded to all event modules using an iterator. An event module keeps track of all pattern instances. Because each event could be the start of a new match, a new pattern instance is created by calling method `createPatternInstance` of class `Machine`. By using an iterator, each pattern instance will try to handle the event.

A pattern instance contains state for a machine. It will forward most methods to the current node of the machine. If, after handling the event, the pattern instance reached a failing state (a state with no outgoing transitions), the instance is terminated by removing it from the list. If the current node of a pattern instance reaches an accepting state, an advice message is delivered to the event filters.

After an event is handled by all pattern instances of all event modules that are imposed on the object, control is returned to the caller of the instrumentation hook.

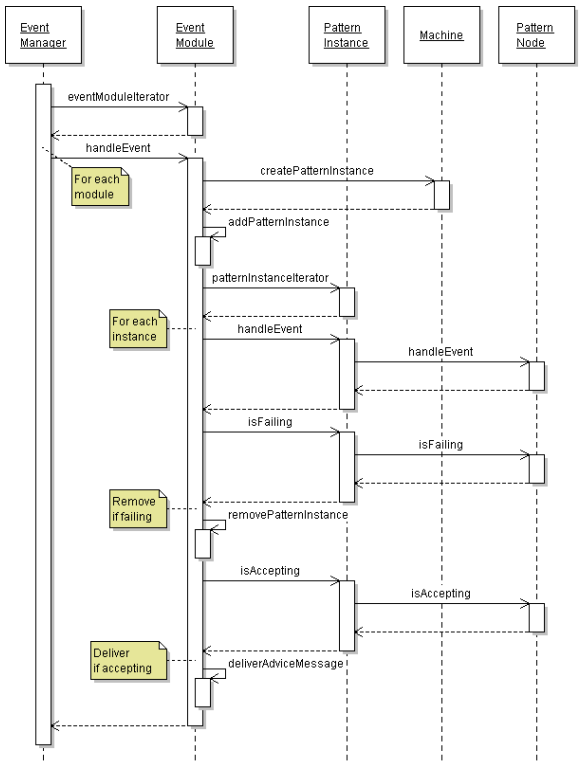


Figure 7.19: Sequence diagram for handling events at run-time

Inside Pattern Instance

When a pattern instance receives an event from an event module, it forwards the request to the current pattern node. This process is shown in the sequence diagram of Figure 7.20. The pattern node will compare the event with the symbol of each outgoing transition. If a symbol uses a variable, the value bound to the current pattern instance is retrieved. If the comparison is successful, the next node is retrieved from the transition and eventually returned by `handleEvent`. For symbols that contain an assignment operator, the value of the variable is set to the value of property from the event. Before returning the next node after a successful comparison, the actions associated to the next node are executed first. Counter values are retrieved through the pattern instance, and its value is increased.

When method `handleEvent` returns control to the event module, a call to `isFailing` and `isAccepting` checks the new current node for its type. A node is failing or accepting when the corresponding properties are set. In addition to checking the property, method `isAccepting` evaluates each condition attached to the node. A condition of class `PatternConditionCompareCounters` will retrieve a set of counters from the pattern instance and compare them. The evaluation is successful when each counter from the set contains the same value and fulfils a certain minimum value (i.e., ≥ 0 or > 0).

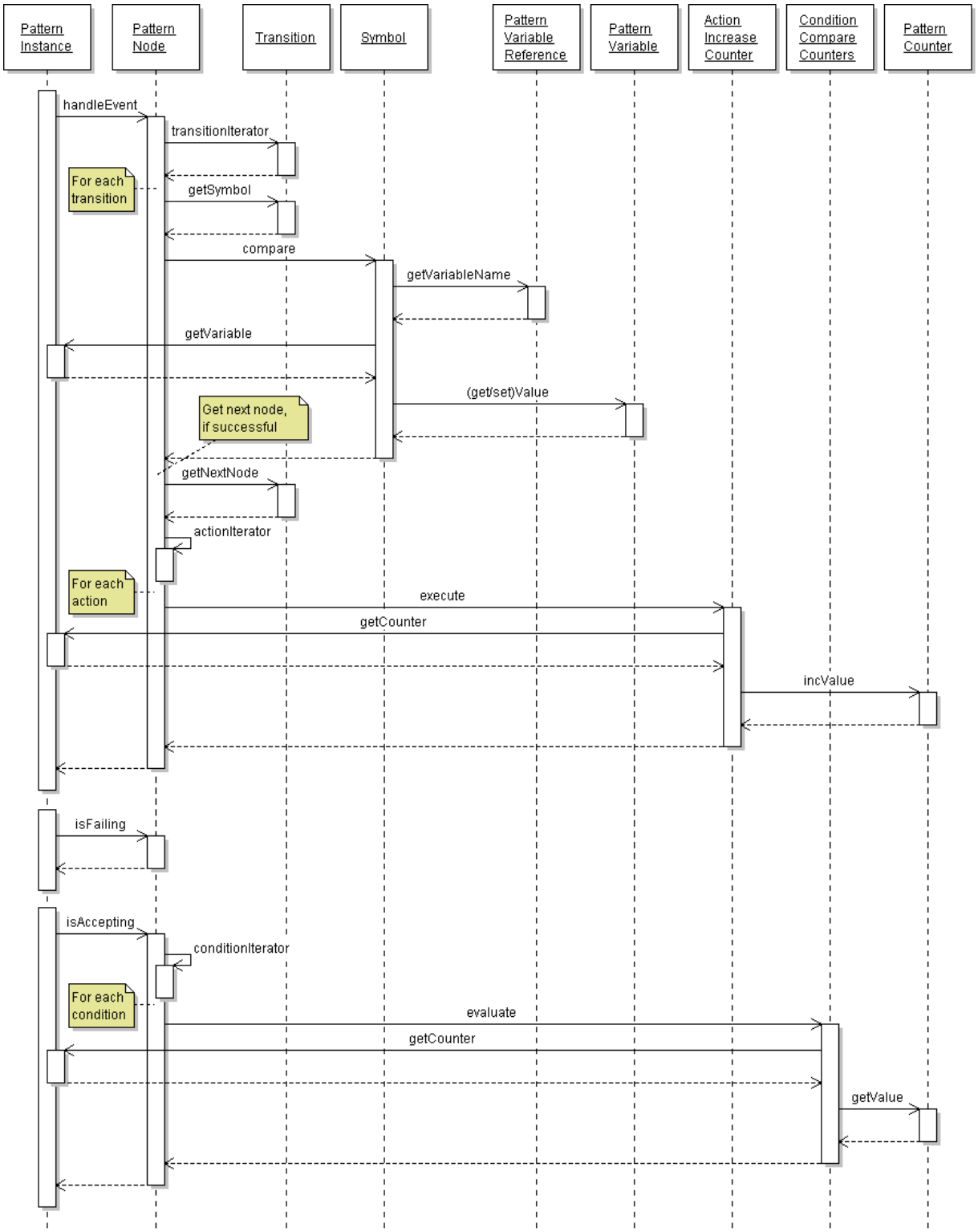


Figure 7.20: Sequence diagram for matching events in pattern instances

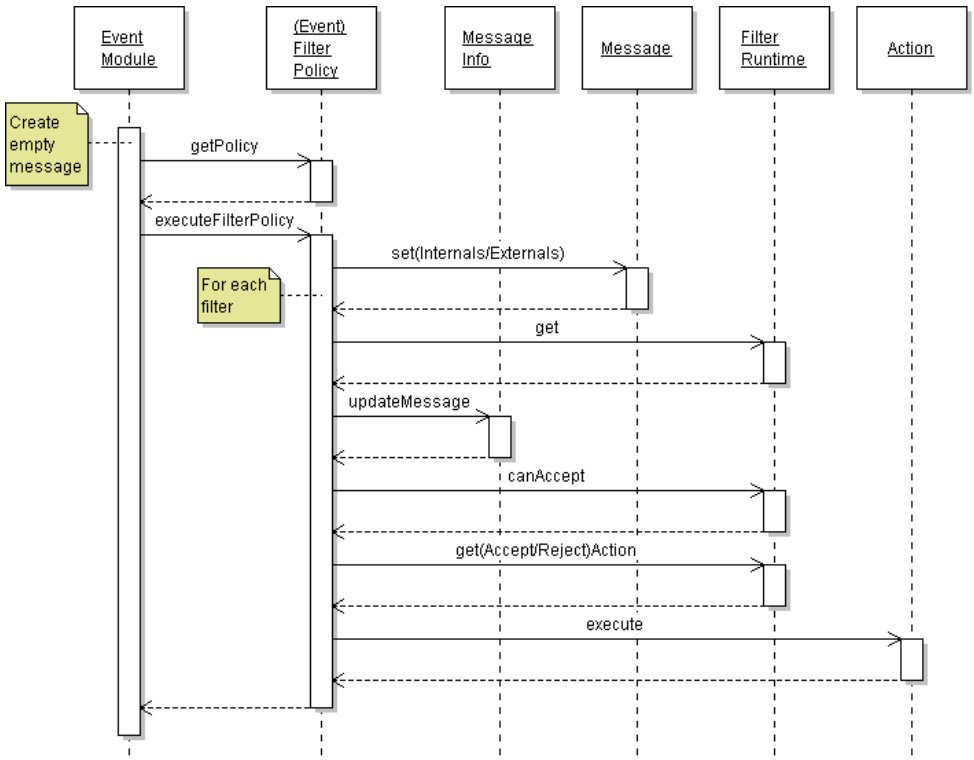


Figure 7.21: Sequence diagram for filtering an advice message in an event module

Delivering an Advice Message

When a pattern reaches an accepting node, some advice is executed as specified by event filters. A sequence diagram for delivering an advice message is depicted in Figure 7.21. The execution policy for event filters has many similarities with the one for message filters (shown in Figure 7.17).

First, an empty message is created with its internals and externals set to those defined in the event module. The default filter policy for event filters is `EventFilterPolicy`. Based on the evaluation of conditions, inside method `canAccept`, the message object is updated with a target and selector. Contrary to the `MetaFilterPolicy` for message filters, each action is executed inside the filter loop. When filter evaluation is finished, control is returned to the calling event manager.

CHAPTER 8

Conclusion

This chapter explains how the work of this thesis relates to similar approaches, followed by a brief overview of the work that has been presented in this thesis. Finally, some perspectives for future work are given.

8.1 Related Work

Various approaches for event-based AOP techniques have been proposed in the past. Two related approaches, that inspired the work of this thesis, are discussed next: event-based AOP and tracematches with free variables.

8.1.1 Event-based AOP

In 'A formal definition of crosscuts', Douence et al. [2001] present an operational model for crosscut definitions based on execution monitors. In this report, join points are represented as events emitted during program execution, and crosscuts are formed by patterns of events, on a match the base program is suspended and advice may be executed. The model is called event-based aspect-oriented programming (EAOP). The language is based on a regular expressions in formulating crosscuts. A basic stateful aspect definition is of the form $C \blacktriangleright I$, where C is a crosscut (pointcut) and I an insert (advice). Aspects can be composed using recursion, sequence and choice. Weaving in EAOP is a parallel composition of aspects.

The formal model allows for proving certain aspect properties, like conflict detection when composing aspects, by taking into account all possible sequences of join points. Aspects are represented as finite state automata, accepting all allowed sequences of events. The base program abstracted into a graph, where program points are nodes and edges represent instructions (and thus events). That graph is instrumented and forbidden sequences will generate an exception (in case of aspects defined as safety properties). After instrumentation, the graph

```
tracematch(Inventory i, ArrayList stock, InventoryDisplay id) {
  sym create_id after returning(id): call(InventoryDisplay.new(..)) && args(i);
  sym change_i after: within(i) && execution(* *(..)) &&
    ((call(* Add(..)) || call(* Remove(..))) && target(stock));

  create_id change_i+
  {
    id.DisplayInventory();
  }
}
```

Listing 8.1: Inventory example for state changing methods using tracematches

is optimized (through minimization and useless state removal) and translated back into an executable program [Douence et al., 2005].

The key difference between EAOP and our approach is the mechanism of monitoring. EAOP uses global monitoring, where event patterns can reason about all events occurring in the program. While global monitoring allows modelling object interactions, the downside is a tight coupling between patterns and the base program. Another difference is the syntax, the pattern language of EAOP allows for multiple inserts in one aspect. In the language presented in this thesis, the pattern has to be broken up in separate event modules, one for each advice. Although conditions can be used to selectively execute certain advice. The pattern language of EAOP is regular, our language extends regular expressions with counters to be able to balance events.

8.1.2 Tracematches with Free Variables

Allan et al. [2005] extended AspectJ with history-based pointcuts, called *tracematches*. A tracematch is specified as a regular expression over events in the program trace. Advice is executed whenever a pattern matches. Events in tracematches correspond to the join point model of AspectJ. The symbols of a pattern are declared using the same syntax as a pointcut in AspectJ. Tracematches not only match events based on type, but also by the values bound to free variables. This allows a tracematch to select those events that are relevant to certain objects instances.

An example tracematch is shown in Listing 8.1. It shows the inventory example used throughout this thesis. The tracematch declares two events of interest in lines 2–4. The first symbol, `create_id`, depicts the creation of an instance of `InventoryDisplay`, whose value is bound to free variable `id`. The subject, an instance of `Inventory`, is passed as an argument and bound to free variable `i`. The second symbol, `change_i`, depicts the execution of a state changing method on inventory `i`. The pointcut of the second symbol designates a call to `Add` or `Remove` on free variable `stock` within class `Inventory`. The pattern of the tracematch is defined on line 6, followed by the advice in parenthesis. It matches after each execution of a state changing method, once a new inventory display is created. On a successful match, method `DisplayInventory` is called. A tracematch allows for multiple sets of bindings for any trace, and thus executes it advice for all

matching bindings. For the observer example, this means we do not need to explicitly store a list of observers for each subject. A consequence of matching on any combination of free variables is that the advice is executed for any variable of type `ArrayList` declared within class `Inventory`. I do not know if it is possible to restrict the pointcut to just private field `stock`.

Tracematches, like EAOP, makes use of a global event monitor. The variable bindings allow for modelling object interactions. Comparing this with our solution to the inventory example, as presented in Section 6.2, we can identify a few differences. Our approach uses local monitoring, which does expose events, like object creation of observers, to other objects than the local one. In our solution uses two filter modules and one event pattern. One filter module is for registration of observers at a subject, the other for registering newly created observers. An event module defines the pattern for detecting state changes.

Patterns in tracematches are specified by regular expressions. In our approach we extended the language with counters, which allows for proper nesting of events. Events in tracematches, however, correspond to pointcuts in AspectJ. This allows them to use the pointcut designators, like `cflow` and `within`, to model nested join points. These constructs allow for limited support for nesting.

8.2 Conclusions

In Chapter 4 the join-point model of Compose* is extended with extra behavioral join points. These join points are represented as events. An event monitor matches these events against pre-defined patterns. During the execution of a program, events are signalled by hooks and observed by event monitors.

Arranged Patterns and Jumping Aspects

The fine-grained join point model allows for selection of join points beyond the level of object interfaces. Using a pattern language as pointcut designator enables context-sensitive crosscuts. These two features contribute to solving the *arranged patterns* and *jumping aspects* problem discussed in Chapter 3. Arranged patterns are solved by abstracting from the name of methods. In the example of Section 6.2, state changing methods are identified by those methods that call certain methods on a private variable. Jumping aspects are solved by defining a context-sensitive crosscut. In the inventory example, a pattern was defined as the execution and return of a state changing function, which matches on those methods that either do a single update or a bulk update. By defining a pattern which matches on a series of updates, the notify advice can, for example, be executed on every fifth state change.

Pattern Language

Chapter 6 presents a notation that integrates event patterns within the Compose⋆ language. Designing a language is always a trade-off between certain quality factors, like expressiveness versus simplicity. Expressiveness is about being able to express a wide range of domain problems in the pattern language. Simplicity is about having clear consistent semantics which are easy to learn. We want to stay as close as possible to the current concepts and syntax of the Compose⋆ framework without sacrificing usability in terms of readability of concern specifications.

A language based on regular expressions is chosen because the language is widely used in a lot of different tools and programming languages. The language contains only a few operators, but nonetheless, allows for a compact representation of complex patterns. Two alternative models are discussed in Section 5.4.3. The explicit fail model corresponds to extended regular expressions by explicitly stating which events are not allowed to happen at that position. The implicit fail model assumes that events corresponding to unused symbols are not allowed to occur anywhere in the pattern. The pattern is described only in terms of events that are allowed to happen. This improves readability of patterns for a small decrease in expressiveness.

Symbol declarations, variables and event module parameters share some of their syntax with the proposal discussed in Section 6.1 for a new filter syntax. For invoking advice, event filters are used to filter an advice message. The event filters are similar to message filters in how they are evaluated.

We took the observer design pattern as an example for the pattern language. Evaluating whether we have the right balance in quality factors can only be done after more domain problems are analyzed.

Local Monitoring

By using local monitoring, concerns remain *strongly encapsulated*. Crosscutting concerns can only reason about the implementation details of their imposed objects, they cannot reason about implementation details of other objects. When a (concern) object is refactored, only those concerns that interact with the modified object might need modifications. For modifications on the public interface of a (concern) object, not only the imposed concerns, but also the output filters of other concerns, can be affected. For modifications on the internal implementation parts, only locally imposed event patterns can be affected. The pattern language, however, is made expressive enough to abstract from a specific value of a property by matching on equality between event properties. Well-written patterns are defined in terms of events that contribute to the goal without depending on a specific value of a property.

Composing Event Modules

Event patterns can be decoupled from advice by making use of *abstract* event modules. This allows for independent reuse of both advice and pattern definitions. In addition, event modules can be composed alongside filter modules. Events are generated from the implementation parts of internals, externals and inner objects. Also, the language is flexible enough to introduce more types of events for reasoning about the process of message filtering within the run-time environment of Compose⋆. As long as the run-time can be instrumented to generate those events. Reasoning about filter processing, allows for concerns on concerns. Also, advice that is executed on a successful match of a pattern is a message delivered to the message handling facility. This makes the message available for further processing by filter modules.

Event modules that are imposed on the same object, might have ordering problems. When the patterns of both event modules match at the same shared event join point, the order between their advice execution might be relevant. The constraint model, discussed in Section 6.4, allows for the specification of ordering constraints inside the superimposition part of a concern. The conflict, however, might not be at a shared join point. For example, when two pieces of advice should not execute if the other advice was executed previously. As events are matched against an execution trace, a proposal is made to add extra constraint relations reasoning about the state of the matching process. An example of such a relation is testing if the automaton, related to the pattern of another event module, is in an accepting state or not.

Design

Chapter 7 explained how event modules can be integrated in the architecture of Compose⋆. It described an architecture for storing event modules inside the repository and construction mechanisms of minimal automata from pattern expressions. An overview is given of the run-time structure of Compose⋆ for instrumenting the message handling facility with events. This allowed the weaver to remain unmodified. It does, however, introduce extra overhead at run-time, as each message is checked for presence of event modules at the sender and receiver.

The complex architecture of the run-time required several workarounds to instrument certain types of *after events*. Filter actions related to meta and custom filters remain to be instrumented. They are executed inside a thread different from the base program.

An overview of the architecture for monitoring was described last. It explained the monitoring process and how advice is executed upon a successful match.

8.3 Future Work

This section presents several future work and enhancements to adding a fine-grained join point model in the Compose⋆ framework.

Local Monitoring with Compound Events

One of the advantages of global monitoring is being able to create a pattern consisting of a series of object interactions. Local monitoring does not allow such patterns, as each instance of an event module is imposed on only one object. Only from that object will it receive events, limiting interaction to incoming messages and outgoing messages. To model larger object interactions, an extension can be made to emit a *compound event* on a successful match of a pattern. By grouping the objects of a program in some kind of hierarchy, these compound events can be sent up a chain of objects. This enables patterns to depend on other patterns without relying on the implementation details of an object.

Pattern Matching using Term Rewriting

The architecture presented in this thesis is based on pattern matching by using final state automata with memory for counters. Another approach is to use a term rewriting system. The term rewriting system can use structural operational semantics to reduce the pattern until it reaches the empty string. This avoids the construction of automata.

Implementation

To successfully implement the design presented in this thesis, a few issues need to be resolved. The current weaver of Compose⋆ does not support all the join points presented in Section 4.3. To support the weaving process, a weave specification needs to be generated from each event pattern.

The number of weave points can be reduced by a applying control flow analysis. Each symbol in a pattern occurs in a certain context. If, at compile-time, it can be determined that an event from a weave point always fails to match against each pattern, that point does not need instrumentation.

The event-based solution to inventory example that is presented in Section 6.2 uses event module parameters. For filter modules, parameters have not been implemented yet. Once implemented, it can be reused for event modules.

To signal events for meta and custom filter actions, a work around needs to be created to instrument them inside the run-time. These actions are executed inside the filter policy thread and on execution they also start their own thread.

The constraint model discussed in Section 6.4.2 can be extended for event modules. Ordering constraints, however, are the only constraints currently implemented in Compose⋆.

Bibliography

M. Akşit (editor). *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*. ACM Press, March 2003.

C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *Adding trace matching with free variables to AspectJ*. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364. ACM Press, 2005. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094839.

American Heritage. *event*. In *The American Heritage, Dictionary of the English Language*. Houghton Mifflin Company, fourth edition, 2004.

L. Bergmans. *Composing Concurrent Objects*. Ph.D. thesis, University of Twente, 1994. URL http://trese.cs.utwente.nl/publications/paperinfo/bergmans.phd.pi.top.htm.

S. R. Boschman. *Performing transformations on .NET Intermediate Language Code*. Master's thesis, University of Twente, The Netherlands, August 2006.

R. Bosman. *Automated Reasoning about Composition Filters*. Master's thesis, University of Twente, The Netherlands, November 2004.

J. Brichau, W. De Meuter, and K. De Volder. *Jumping Aspects*. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr (editors), *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*. June 2000. URL http://trese.cs.utwente.nl/Workshops/adc2000/papers/Brichau.pdf.

R. Casati and A. Varzi. *Events*. In E. N. Zalta (editor), *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab at the Center for the Study of Language and Information, Stanford University, fall 2002 edition, 2000. URL http://plato.stanford.edu/archives/fall2002/entries/events/.

R. Cleaveland and O. Sokolsky. *Equivalence and preorder checking for finite-state systems*. In J. A. Bergstra, A. Ponse, and S. A. Smolka (editors), *Handbook of Process Algebra*, pages 391–424. Elsevier Science, 2001. ISBN 0-444-82830-3.

O. Conradi. *Fine-grained Join Point Model in Compose**. Master's thesis, University of Twente, The Netherlands, August 2006.

D. Doornenbal. *Analysis and Redesign of the Compose* Language*. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

R. Douence, P. Fradet, and M. Südholt. *Trace-Based Aspects*. In [Filman et al., 2005], pages 201–217.

R. Douence, O. Motelet, and M. Südholt. *A Formal Definition of Crosscuts*. In A. Yonezawa and S. Matsuoka (editors), *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 170–186. Springer-Verlag, September 2001.

P. Dürr, L. Bergmans, G. Gülesir, and I. Nagy. *Towards an Expressive and Scalable Framework for expressing Join Point Models*, 2005. URL http://trese.cs.utwente.nl/publications/files/0419FOAL_AOSD2005.pdf. Submitted to the FOAL 2005 workshop in Chicago, IL, USA.

P. E. A. Dürr. *Detecting semantic conflicts between aspects (in Compose*)*. Master's thesis, University of Twente, The Netherlands, April 2004.

T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming*. In *Comm. ACM*, volume 44(10):pages 29–32, October 2001.

R. E. Filman, T. Elrad, S. Clarke, and M. Akşit (editors). *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

R. E. Filman and D. P. Friedman. *Aspect-Oriented Programming Is Quantification and Obliviousness*. In [Filman et al., 2005], pages 21–35.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

C. F. N. García. *Compose* - A Runtime for the .NET Platform*. Master's thesis, Vrije Universiteit Brussel, Belgium, August 2003.

D. Giammarresi, J.-L. Ponty, D. Wood, and D. Ziadi. *A characterization of Thompson digraphs*. In *Discrete Appl. Math.*, volume 134(1-3):pages 317–337, 2004. ISSN 0166-218X. doi:http://dx.doi.org/10.1016/S0166-218X(03)00299-3.

D. Giannakopoulou and K. Havelund. *Automata-Based Verification of Temporal Properties on Running Programs*. In *ase*, volume 00:page 412, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2001.989841.

M. Glandrup. *Extending C++ using the concepts of Composition Filters*. Master's thesis, University of Twente, 1995. URL http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm.

J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. ISBN 0471431044.

K. Gybels and J. Brichau. *Arranging Language Features for Pattern-based Crosscuts*. In [Akşit, 2003], pages 60–69.

K. Havelund and G. Roșu. *Efficient monitoring of safety properties.* In *Int. J. Softw. Tools Technol. Transf.*, volume 6(2):pages 158–173, 2004. ISSN 1433-2779. doi:http://dx.doi.org/10.1007/s10009-003-0117-6.

W. Havinga. *Designating join points in Compose* - a predicate-based superimposition language for Compose*.* Master's thesis, University of Twente, The Netherlands, May 2005.

E. Hilsdale and J. Hugunin. *Advice weaving in AspectJ.* In K. Lieberherr (editor), *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 26–35. ACM Press, March 2004. doi:http://doi.acm.org/10.1145/976270.976276.

F. J. B. Holljen. *Compilation and Type-Safety in the Compose* .NET Environment.* Master's thesis, University of Twente, The Netherlands, May 2004.

R. L. R. Huisman. *Debugging Composition Filters.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

S. H. G. Huttenhuis. *Patterns within Aspect Orientation.* Master's thesis, University of Twente, The Netherlands, 2006. To be released.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ.* In J. L. Knudsen (editor), *Proc. ECOOP 2001, LNCS 2072*, pages 327–353. Springer-Verlag, Berlin, June 2001.

P. Koopmans. *Sina user's guide and reference manual.* Technical report, Dept. of Computer Science, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/sinaUserguide.pi.top.htm`.

O. Kupferman, N. Piterman, and M. Vardi. *Extended Temporal Logic Revisited.* In *12th International Conference on Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 519–535. August 2001.

T. Leslie. *Efficient Approaches to Subset Construction.* Master's thesis, University of Waterloo, Ontario, Canada, 1995.

B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997. ISBN 0-13-629155-4.

I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* Ph.D. thesis, University of Twente, The Netherlands, June 2006.

G. van Noord. *Treatment of Epsilon Moves in Subset Construction.* In *Computational Linguistics*, volume 26, pages 61–71. MIT Press, Cambridge, MA, USA, 2000. URL `http://citeseer.ist.psu.edu/vannoord00treatment.html`.

H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In M. Akșit (editor), *Software Architectures and Component Technology.* Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

M. D. W. van Oudheusden. *Automatic Derivation of Semantic Properties in .NET*. Master's thesis, University of Twente, The Netherlands, August 2006.

A. Pnueli. *The temporal logic of programs*. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

A. Popovici, G. Alonso, and T. Gross. *Just in Time Aspects*. In [Akșit, 2003], pages 100–109.

A. Popovici, T. Gross, and G. Alonso. *Dynamic Weaving for Aspect-Oriented Programming*. In G. Kiczales (editor), *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, April 2002.

P. Salinas. *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*. Master's thesis, Vrije Universiteit Brussel, August 2001.

D. R. Spenkelink. *Compose* Incremental*. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

T. Staijen. *Towards Safe Advice: Semantic Analysis of Advice Types in Compose**. Master's thesis, University of Twente, April 2005.

T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, second edition, 1997. ISBN 0-201-82136-2.

P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In [Filman et al., 2005], pages 37–61.

C. Vinkes. *Superimposition in the Composition Filters Model*. Master's thesis, University of Twente, The Netherlands, October 2004.

D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

J. C. Wichman. *The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`.

J. W. te Winkel. *Bringing Composition Filters to C*. Master's thesis, University of Twente, The Netherlands, 2006. To be released.