Applying Composition Filters to C

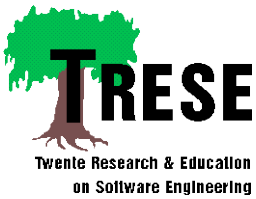
A thesis submitted for the degree
of Master of Science at
the University of Twente

J.W. te Winkel

Enschede, December 15, 2006

Graduation commission:
Prof. dr. ir. M. Akşit
Dr. ir. L.M.J. Bergmans
Ir. P.E.A. Dürr

Twente Research and Education
on Software Engineering
Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente



University of Twente
Enschede - The Netherlands

Summary

Aspect-oriented programming makes it possible to separate crosscutting concerns from the base program. It is an extension to the object-oriented programming paradigm, but crosscutting concerns are not an object-oriented problem. Such concerns are also seen in other programming languages, for instance in C.

Bringing aspect-oriented programming to C is valuable, because C is still very popular and crosscutting concerns are troublesome. This is for example the case in the legacy system of ASML, where four major crosscutting concerns are responsible for approximately 31% of the 12 million lines of code in the system.

At the moment there are a few aspect-oriented weavers for C. These weavers are all based on the AspectJ methodology. In this research an aspect-oriented weaver for C is developed with the Composition Filters approach. The main benefit that composition filters offer over the AspectJ approach is that it supports a declarative way for specifying advice by use of a filter.

For implementing a weaver, first a mapping from composition filters to C is developed. This is not easy, because composition filters are an extension to the object-based paradigm, and it uses some language constructs from this paradigm which do not exist in C. Key elements in composition filters are: a message, a filter, and a superimposition unit. In order to create a weaver for C, these elements need to be defined in C.

After the mapping is created, Compose⋆ C is presented, which is an extension of the current implementation of composition filters named Compose⋆. Extending Compose⋆ was possible, because a large part of its design and implementation is language independent which means that it could be reused. Components that are not language independent are: the code-weaver, the language model, the collecting of program-information, the filters, and IDE support. Therefore these components had to be developed by ourselves, except for the code-weaver for which an existing weaver is selected.

Acknowledgements

This research is executed for the graduation of the Computer Science study at the University of Twente. I performed this research at the Twente Research and Education on Software Engineering chair, under the supervision of Lodewijk Bergmans and Pascal Dürr.

I attained this assignment, because I did a similar assignment at Océ Technologies in Venlo as an internship. In this assignment an evaluation was made of the use of aspect-oriented programming for Océ. This assignment was also supervised by Lodewijk Bergmans, so when the internship was nearly finished and I contacted Lodewijk about a graduation assignment, he directly came up with this one.

With this chapter I want to thank all the people who helped me throughout my research. First of all special thanks to my two supervisors, Lodewijk Bergmans and Pascal Dürr, who of course came up with this graduation assignment, but also gave me valuable input, guided me when ever I was stuck, and helped me with my thesis.

Secondly, I want to thank my project room members, Arjan de Roo, Dirk Doornenbal, Michiel Hendriks, Olaf Conradi, Rolf Huisman, Marcus Klimstra, and Stephan Huttenhuis, for all the time they spend helping with problems, showing me the options of Compose⋆, and working together on the common chapters. Together, with Roy Spenkelink I developed the Eclipse plugin, also thanks to him. Thanks to Wilke Havinga for the Prolog/ Language model support he offered me. And last, I want to thank all the other Compose⋆ developers in general, for all the valuable discussions over Compose⋆ and the pleasant time during this project.

Johan Willem te Winkel
December 15,2006
Enschede, The Netherlands

Contents

List of Figures

Listings

Chapter 1

Introduction to AOSD

The first two chapters have originally been written by seven M. Sc. students [38, 28, 60, 15, 57, 37, 14] at the University of Twente. The chapters have been rewritten for use in the following theses [59, 23, 58, 40, 24, 39, 55]. 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 [61]. 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 values from input values. Prolog is currently the most used logic language [61].

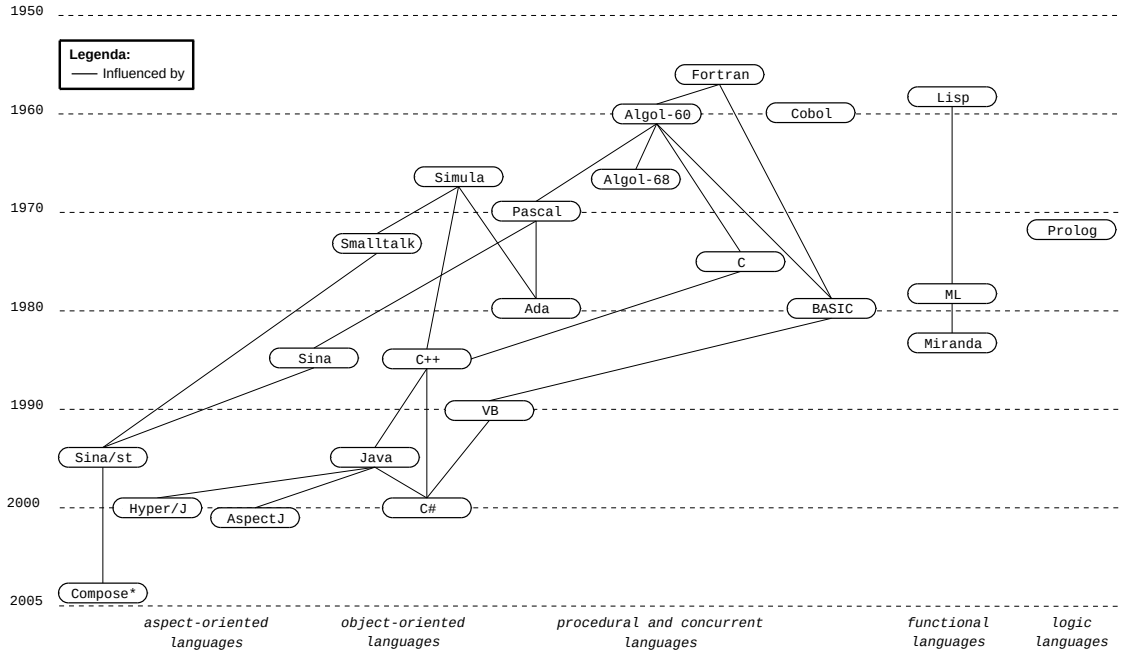


Figure 1.1: Dates and ancestry of several important languages

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 [61]. 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 [34].

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* [32]. 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 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, 15, and 22 in Listing 1.1a and 2, 5, and 10 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.

```java
public class Add extends Calculation{

  private int result;
  private CalcDisplay calcDisplay;
  private Tracer trace;

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

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

  public int getLastResult() {
    trace.write("int
        Add.getLastResult()");
    return result;
  }
}
```

(a) Addition

```java
public class CalcDisplay {
  private Tracer trace;

  public CalcDisplay() {
    trace = new Tracer();
  }

  public void update(int value){
    trace.write("void
        CalcDisplay.update(int)");
    System.out.println(
        "Printing new value of
        calculation: " +value);
  }
}
```

(b) CalcDisplay

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

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

(a) Addition concern

```
1  aspect Tracing {
2   Tracer trace = new Tracer();
3
4   pointcut tracedCalls():
5    call(*(Calculation+).*(..))
6    ||
7    call(*CalcDisplay.*(..));
8
9   before(): tracedCalls() {
10   trace.write(thisJoinPoint.
11    getSignature().
12    toString());
13  }
14 }
```

(b) Tracing concern

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

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 [36]. 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;
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 subsubsection 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 subsubsection 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [48, 49].

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. [29], 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 [43] 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;

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

```
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

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.

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

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

Listing 1.4: Example of static crosscutting in AspectJ

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

Listing 1.5: Creation of a hyperspace

separation of concerns [46], 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.

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

Listing 1.6: Specification of concern mappings

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

Listing 1.7: Defining a hypermodule

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, and Java.

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 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 two target environments: the .NET, and Java. 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 [44].

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 [10].

1995 The Sina language with Composition Filters is implemented using Smalltalk [44]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [35].

1999 The composition filters language ComposeJ [62] 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 [52]. 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⋆.

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, and annotations are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

```
1  concern {
2    filtermodule {
3      internals
4      externals
5      conditions
6      inputfilters
7      outputfilters
8    }
9
10   superimposition {
11     selectors
12     filtermodules
13     annotations
14   }
15
16   implementation
17 }
```

Listing 2.1: Composition Filters concern elements

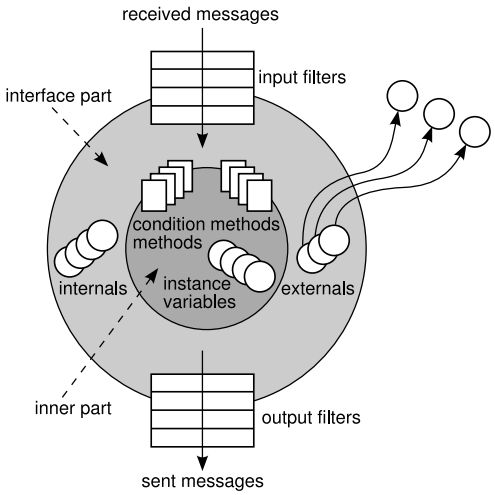


Figure 2.1: Components of the composition filters model

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 built-in 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.

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,

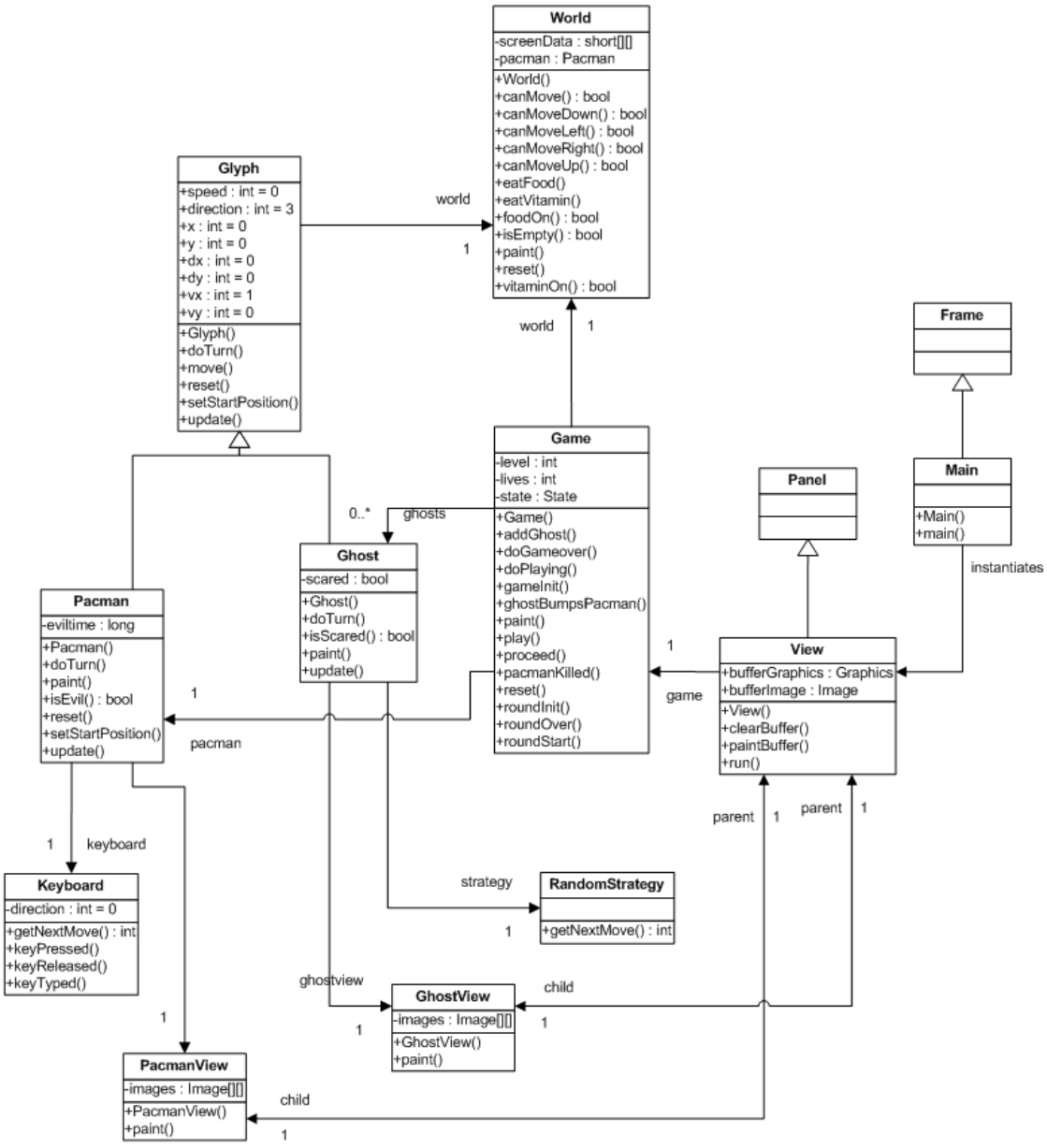


Figure 2.2: Class diagram of the object-oriented Pacman game

```
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⋆

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.

```
1  public class Score
2  {
3    private int score = -100;
4    private static Score theScore = null;
5    private Label label = new java.awt.Label("Score: 0");
6
7    private Score() {}
8
9    public static Score instance() {
10     if(theScore == null) {
11       theScore = new Score();
12     }
13     return theScore;
14   }
15
16   public void initScore(ReifiedMessage rm) {
17     this.score = 0;
18     label.setText("Score: "+score);
19   }
20
21   public void eatGhost(ReifiedMessage rm) {
22     score += 25;
23     label.setText("Score: "+score);
24   }
25
26   public void eatVitamin(ReifiedMessage rm) {
27     score += 15;
28     label.setText("Score: "+score);
29   }
30
31   public void eatFood(ReifiedMessage rm) {
32     score += 5;
33     label.setText("Score: "+score);
34   }
35
36   public void setupLabel(ReifiedMessage rm) {
37     rm.proceed();
38     label = new Label("Score: 0");
39     label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
40     Main main = (Main)Composestar.Runtime.FLIRT.
41         message.MessageInfo.getMessageInfo().getTarget();
42     main.add(label,BorderLayout.SOUTH);
43   }
44 }
```

Listing 2.3: Implementation of class `Score`

```
1  concern DynamicStrategy in Pacman {
2    filtermodule dynamicstrategy {
3      internals
4        stalk_strategy : pacman.Strategies.StalkerStrategy;
5        flee_strategy : pacman.Strategies.FleeStrategy;
6      conditions
7        pacmanIsEvil : pacman.Pacman.isEvil();
8      inputfilters
9        stalker_filter : Dispatch = {!pacmanIsEvil =>
10                         [*.getNextMove] stalk_strategy.getNextMove};
11       flee_filter : Dispatch = {
12                         [*.getNextMove] flee_strategy.getNextMove}
13   }
14   superimposition {
15     selectors
16       random = { C | isClassWithName(C,
17                         'pacman.Strategies.RandomStrategy') };
18     filtermodules
19       random <- dynamicstrategy;
20   }
21 }
```

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 [45]: IDE, compile-time, adaptation, and runtime.

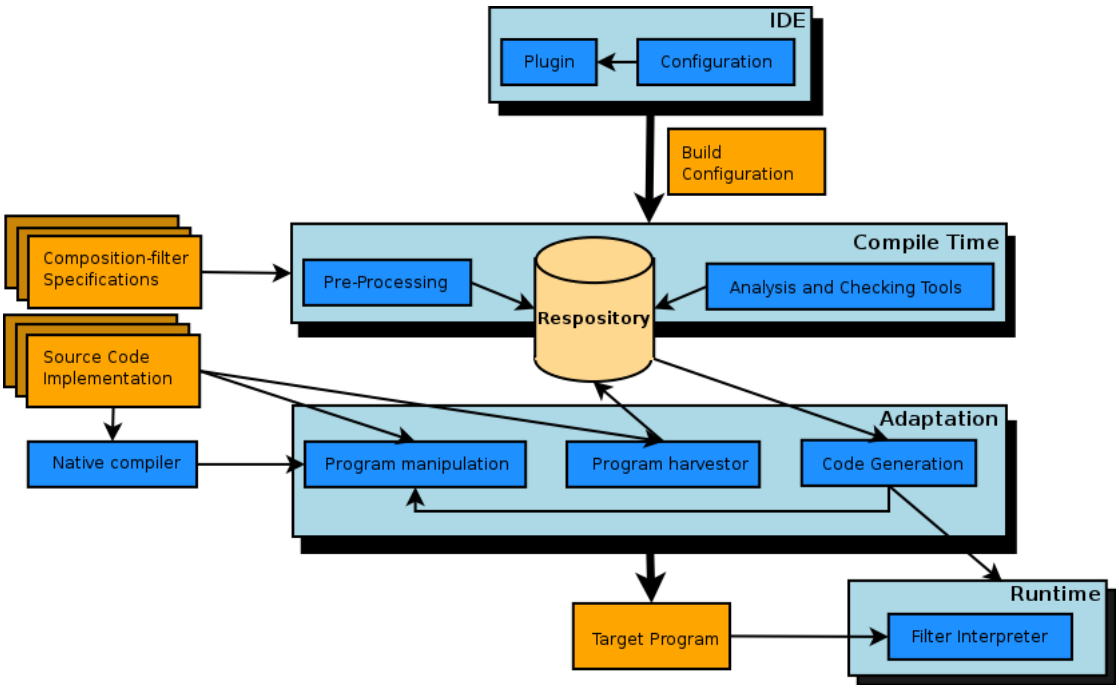


Figure 2.3: Overview of the Compose⋆ architecture

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. For example Compose⋆.NET supports a plug-in for Visual Studio. 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 knowledge-base, called the 'repository'. This

knowledge-base contains the structure and meta-data 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 validation 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 knowledge-base. During code generation a reduced copy of the knowledge-base 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 knowledge-base, 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 two platforms: .NET, and Java. 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⋆ Java targets the Java platform, but is still under construction.

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 filter modules 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 provide a plug-in for an IDE. Compose⋆.NET provides a plug-in for Visual Studio, while the objective of Compose⋆ Java is to 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

Problem Identification

This chapter will motivate why we want to apply composition filters to C. It starts with a description of problems that are caused by crosscutting concerns, which is then clarified with a concrete example. After this the choice for composition filters is explained. This chapter results in some requirements for the prototype of Compose⋆ C.

3.1 Background

Aspect-oriented programming makes it possible to split crosscutting concerns from the base code, by placing them in a new module called an aspect. Splitting crosscutting concerns improves the modularity of the program, because this has a positive influence on the reusability, changeability and understandability of the programming code, Section 1.2. Aspect-oriented programming is developed as an extension to the object-oriented paradigm, but recent research has shown that crosscutting concerns are not an object-oriented problem. One of the not-object-oriented languages where problems with crosscutting concerns are identified is C, which is the focus of this research.

C is a procedural programming language, which means that it has less support for modularity techniques. Problems with such languages were already recognized years ago and resulted in new programming paradigms such as object-oriented programming. However, C is still very popular because of its efficiency. Therefore extending C with aspect-oriented programming is valuable, because aspect-oriented programming addresses modularity problems.

Aspect-oriented programming for C can especially be valuable for reducing problems with existing legacy systems. A legacy system can be defined as "any information system that significantly resists modification and evolution"[16]. It is characterized as a large and complex system, with unacceptable high maintenance costs, a result of evolution over years. Nevertheless, in these years such a system has become mission critical,

and therefore it is not an option to stop using the system [12][22]. Such a system is troublesome, when business needs are changing, and therefore the requirements of the system need to change. These changes will result in high costs. Modernizing, or also called re-engineering, a legacy system is required in order to improve its evolvability. An example of a system that rejects evolution, because of problems caused by crosscutting concerns, is described in [17], and [27]. This system consists of 12 million lines of code and contains four major crosscutting concerns that are spread out over the entire program. Together these concerns take up approximately 31% of the code.

In the next section the modularization problems in C that are caused by crosscutting concerns will be explained on the basis of an example.

3.2 Crosscutting Examples

The last few years, recognition of crosscutting problems in C is risen. For example in [54], [26] the following crosscutting concerns are described: memory and thread management, event dispatching and handling, initialization and destruction for efficient start up and secure shutdown, network communication, configuration of middleware, exchangeability, switching the network protocol, buffer overflows and prefetching.

In this thesis the crosscutting concerns that are found by the Ideals project in the legacy system of ASML [17] are central to our research, because these concerns are well documented [18, 27] and give a clear insight of the problems with crosscutting concerns in C.

In the legacy system of ASML four major crosscutting concerns are distinguished. At ASML it is mandatory that these concerns are applied to every function in the system. In Listing 3.1 a function is shown, where the ASML concerns are applied to. These concerns are:

Reflection
: Functions have a local variable that contains the function's name, which is used for logging. This variable is set at the beginning of the function as in line 13;

Dynamic Execution Tracing
: Logs the values of the in-, out-, inout- parameters at the beginning or end of a function for debugging purposes, see lines 29, 36. These functions need a tracing handler that takes care of the tracing. This handler is declared in a global struct, line 5;

Parameter Checking
: Examines if an in-, or inout- parameter is not NULL and if an out-parameter is NULL, if this is the case an error needs to be logged, and also an error needs to be returned to the calling function, lines 14, and 18;

Timing
: Times the function's execution. To do this first a timer handler is declared, line 4

and defined, line 24. This handler is used in the actual timing supported by the in and out functions in lines 27, and 39. [18]

The function in the example divides parameter *op1* by parameter *op2* and writes the result in parameter *answer*, if *op2* is not zero, line 32. This is the only code of the function and it only covers a small part of the total code-size of the function.

```
1  typedef struct
2  {
3    ...
4    TIMING_handle ti_handle;
5    TRACING_handle tr_handle;
6      ...
7  } CALCmodule_data_struct;
8
9  ...
10
11 int CALCdivide(int * answer, int op1, int op2)
12 {
13     const char* func_name = "CALCdivide";
14     if (op1 == NULL || op2== NULL){
15         ERROR_LOG("Input Parameter error in function : \%i",func_name);
16         return(INPUT_PARAMETER_ERROR);
17     }
18     if (answer == NULL){
19         ERROR_LOG("InOutput Parameter error in function: \%i",func_name);
20         return(INOUT_PARAMETER_ERROR);
21     }
22     int result = OK;
23
24     TIMING_handle TIMING_hdl = NULL;
25
26     TIMING_in(TIMING_func_timing_hdl, CALCmodule_data_struct.ti_handle,
27         func_name, &TIMING_hdl);
28
29     TRACE_in( CALCmodule_data_struct.tr_handle, func_name);
30
31     /** original code of function CALCdivide **/
32     if(op2 != 0)
33         *answer = op1 / op2;
34     else result = ERROR;
35
36     TRACE_out(CALCmodule_data_struct.tr_handle, func_name, result);
37
38     TIMING_out(TIMING_func_timing_hdl, CALCmodule_data_struct.ti_handle,
39         func_name, &TIMING_hdl);
40
41     return result;
42 }
```

Listing 3.1: `Function from file Calc.c with all` ASML concerns

The Figures 3.1, 3.2, 3.3, and 3.4 show a component of the ASML system, where occur-

rences of the code belonging to a specific concern are highlighted.

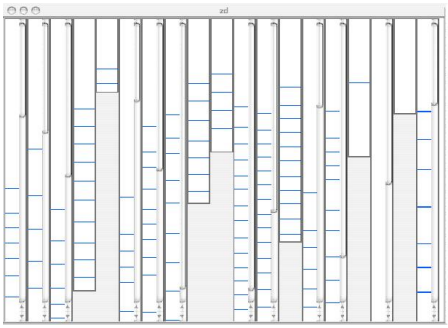


Figure 3.1: Reflection concern occurrences

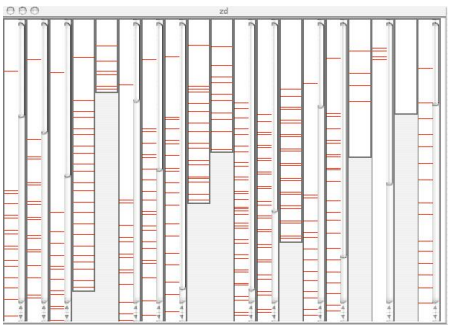


Figure 3.2: Tracing concern occurrences

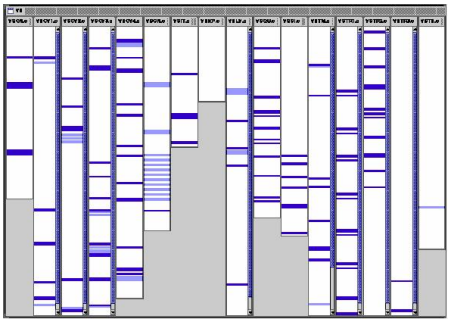


Figure 3.3: Parameter Checking concern occurrences

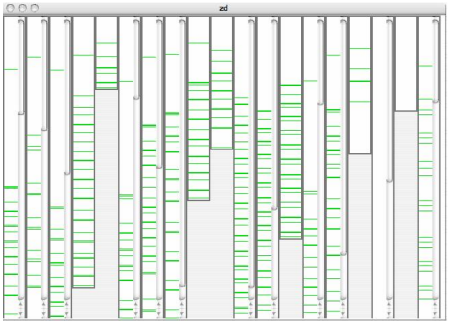


Figure 3.4: Timing concern occurrences

The example and figures show that the problems described in Section 1.2, apply to these concerns. These problems are:

Code is difficult to change
: Because the crosscutting concerns are scattered over the entire program, making changes to one of these concerns may require a change at all the positions in the program where code belonging to the concern is placed. As we see in the Figures 3.1, 3.2, 3.3, and 3.4 this would be a very time consuming job, while these components are only a small part of the entire system. Furthermore, when modifications are made to one of the tangled files, checking for side-effects with all crosscutting concerns is required;

Code is harder to reuse
: When a function of this system needs to be reused, the crosscutting code should be deleted or the code should also be added to the other functions in the new system. Reusing the function with crosscutting code requires the copying of used global variables, such as the handler that is used by the tracing concern, line 5 from Listing 3.1;

Code is harder to understand
Tangled code makes it difficult to see which code belongs to which concern. This problem becomes clear through Listing 3.1, where only lines 32 till 34 belong to the *CALCdivide* function.

3.3 Composition Filters

In the preceding paragraphs the need for aspect-oriented programming in C has been clarified. This was already concluded by some other projects, scuh as: AspectC, AspectC++, Arachne, and WeaveC. These projects all resulted in an aspect-oriented weaver that works and possibly can tackle the problems described in the paragraphs. However, these weavers are based on the AspectJ methodology described in Section 1.4.

The intention of this research is to develop an aspect-oriented weaver for C with the Composition Filters approach. We have chosen for composition filters, because of its declarative manner of specifying advice, by using a filter. Declarative specifying advice is easy to understand, and therefore it is simple to create composition filters concerns. Furthermore, because of the declarative way to specify filters it is possible to reason about the semantics of a concern. Nevertheless, the applicability of composition filters to C is questionable. It is stated that the concept of composition filters is language independent. However, it is developed as an extension of the object-based paradigm.

The implementation of the Composition Filters approach, Compose⋆, uses the reasoning to improve the control over the program, see Section 2.6. This section also shows that Compose⋆ is very extensive. Porting Compose⋆ to C, Compose⋆ C, should make it possible to use some of these features. It seems feasible, because during the design of Compose⋆, support for multiple languages is taken into account. At the moment it already has the ability to weave at IL and Java.

3.4 Compose⋆ C Requirements

Before we start looking how composition filters can be applied to C and how we can realize a weaver, we need to specify some non-functional or quality requirements that the weaver should fulfill. These requirements are needed in order to make design decisions throughout this research. Functional requirements of the weaver are not discussed here because they cannot be defined before we know what the weaver should do.

Requirements can be deduced from the characteristics of C as well as the intention of composition filters. Characteristics of C that need to be taken into account when an aspect-oriented weaver is developed are:

General applicable
It needs to be possible to use the weaver on existing programs. This is not easy,

because there are several C dialects. The working of the prototype should be independent from naming conventions that are common in C projects;

Efficient
A C program is very efficient. One of the requirements we would like to fulfill for this prototype is that the execution overhead stays within acceptable limits, after using the weaver.

Composition filters is a language independent concept, which has resulted in multiple Compose⋆ implementations for different languages. We would like the weaver to fulfill the requirement:

Compatibility of Compose⋆ concerns
Using the same composition filters syntax will make the understanding of composition filters easier and it will make it possible to reuse concerns across platforms.

3.5 Summary

Aspect-oriented programming makes it possible to separate crosscutting concerns from the base code; in this way the modularity of a program will be improved. Aspect-oriented programming for C can be very valuable, because C lacks modularity techniques, and recent research has shown that crosscutting concerns are problematic in C.

There already are some aspect-oriented weavers for C. However, these weavers are all based on the AspectJ approach. In this research an aspect-oriented C weaver based on the Composition Filters approach will be developed, Compose⋆ C. Compose⋆ has some specific, useful features and is easy to port to other languages.

Based on the characteristics of C and an intention of the Compose⋆ project some requirements are formulated. The weaver needs to be generally applicable and preserve the performance of a C program. Also, the Compose⋆ concerns need to be compatible with the other Compose⋆ implementations.

Chapter 4

The C Programming Language

C is a procedural programming language developed in the early 1970s by Dennis Ritchie. It was intended for use on the Unix operating system, but has spread to many others. At the moment it is one of the most popular programming languages. Because a C program is very efficient, C is still very popular for writing system software. However, C can also be used for developing other kind of applications. [19]

4.1 C-Program

A C-program consists of text that is kept in units called source- or preprocessing files. A source file, together with all the headers and source files included via the preprocessing directive #include, is known as a preprocessing translation unit. After preprocessing, a preprocessing translation unit is called a translation unit. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by, for example, calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program. [41] [42]

A C-program is a sequence of functions that call each other for processing. This sequence starts with a function named *"main"*. In C, functions are the way to abstract information. Functions can be reused from the standard library that comes with the compiler, or developed by oneself.

4.2 Type system

The type system of C supports the basic types, char, int, float, double and enumerations, where some qualifiers can be applied to, for instance short and long to integers. From

the basic types, derived types are constructed, being:

Arrays
are sets of objects of a given type;
Functions
encapsulate some computation and hide the details of the operation from other parts of the program, which do not need to know about them;
Pointers
are variables that contain the address of a variable;
Structures
are collections of one or more variables, possibly of different types, grouped together under a single name for convenient handling;
Unions
are variables that may hold objects of different types and sizes. It is a single area of storage big enough to hold the widest member. [41] [42]

Nevertheless C does not support classes.

C is a weak-typed language. The compiler does check if the type of an expression is correct, but it is possible in C to override the checks. In C a value of one type can be treated as another, so it is for instance possible to store a character in a int, because the size of an int is larger than a char. By use of a cast the actual character can be retrieved.

4.3 External objects

A C program consists of a set of external objects, which are either variables or functions. External variables are defined outside of any function and are therefore available to many functions. By default external variables and functions have the property that all references by the same name are references to the same thing. External variables are permanent so they retain values from one function invocation to another. Any function may access an external variable by referring to it by name, if the name has been declared somehow. The scope of an external variable or function is the point from which it is declared, to the end of the file being compiled. If an external variable is used before it is defined, or when it is defined in another source file than it is used, an extern declaration is needed. In C there is a difference between a definition and declaration of an extern variable. A definition sets aside storage and also serves as the declaration for the rest of the file. A declaration announces the properties of a variable, primarily its type. By prefixing an external object with the word static, the scope of the object will be limited to the rest of the source file being compiled. This is a way to hide the objects. [42]

4.4 Local variables

Variables declared in a function are named automatic/local. The scope of such a variable is the function where it is declared. Variables of the same name in different functions are unrelated. With internal variables also the static declaration can be used. Static means here that the local variables remain in existence rather than coming and going each time a function is activated. Internal static variables provide private, permanent storage within a single function.[42]

4.5 Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptually a separate first step in the compilation process. One of the most frequently used features is #include, which makes it possible to include the contents of a file during compilation. There are often several #include lines at the beginning of a source file, to include common #define statements and extern declarations, or to access the function prototype declarations for library functions from headers. Included files may also contain #include lines. #include is the preferred way to tie the declarations together for a large program. [42]

4.6 Naming Conventions

In C it is common to use naming conventions for adding meta-information to your program. With naming conventions the level of abstraction can be enhanced, compared to normal source code; this makes the software for instance better maintainable. Naming conventions are used in [4] and [5] to imitate for instance namespaces, by applying file naming conventions. At ASML naming conventions are used to define modules. However it is not realistic to expect that everyone applies naming conventions with care [4][5] and it is possible that naming conventions overlap other identifier names, for instance using *set** functions for setting a variable and a function *settle*[45].

4.7 Example

In the following listings an example *HelloWorld* C-program is seen. This example prints the two lines, *"Hello World"*, and *"Hello user"*, and shows most of the concepts discussed in this chapter. In line 3 of 4.1 the *main* function is seen, which calls another function, *worldAnswer* line 6. *worldAnswer* is declared in the file *World.h*, 4.4, which is included in line 2. The definition of *worldAnswer* is found in 4.3. In 4.1 line 4 a local variable of type *char** is shown, and in 4.3 line 3 a *char** external variable. In this example also a

naming convention is applied; all elements which belong to the file *World*, header and source file, start with the pattern *"world"*.

```c
#include "HelloWorld.h"
#include "World.h"
int main(){
    char* userText= "Hello World";
    printf("%s \n", userText);
    worldAnswer();
    return 0;
}
```

Listing 4.1: `HelloWorld.c`

```c
#include <stdio.h>
int main();
```

Listing 4.2: `HelloWorld.h`

```c
#include "World.h"

char* worldText ="Hello user";

void worldAnswer(){
    printf("%s\n",worldText);
}
```

Listing 4.3: `World.c`

```c
#include <stdio.h>
void worldsAnswer();
```

Listing 4.4: `World.h`

4.8 Summary

C is an older programming language, but still often used because of its efficiency. A C program is a series of functions that call each other. A function in C is used to abstract behavior, and therefore it is the unit that can be reused. C's type system consists of basic and derived types, but does not support classes. The scope of an object in C can be local or external. Local means that a variable is defined inside the function. Whereas an external object is a function or variable that can be used in the whole file it is defined. An external object can be used in another file when it is declared as extern. Separate files can be combined into a program by using the preprocessor. In C it is common to use naming conventions to enhance the abstraction level.

Chapter 5

Composition Filters in C

With composition filters, messages that enter and exit an object can be intercepted and manipulated by a sequential composition of filters. This makes it possible to modify the behavior of an object. By use of superimposition filter modules can be added to objects [11]. In Figure 5.1 the working of composition filters is shown.

Composition filters are developed as an extension of the object-based model. Object-based languages are higher structured imperative languages than procedural languages, such as C. The intention of this thesis is to apply composition filters to the programming language C. Characteristics that distinguish object-based languages from other imperative languages are:

- The concepts Objects and Classes;
- Objects are first class-values. This means that objects can be used as arguments

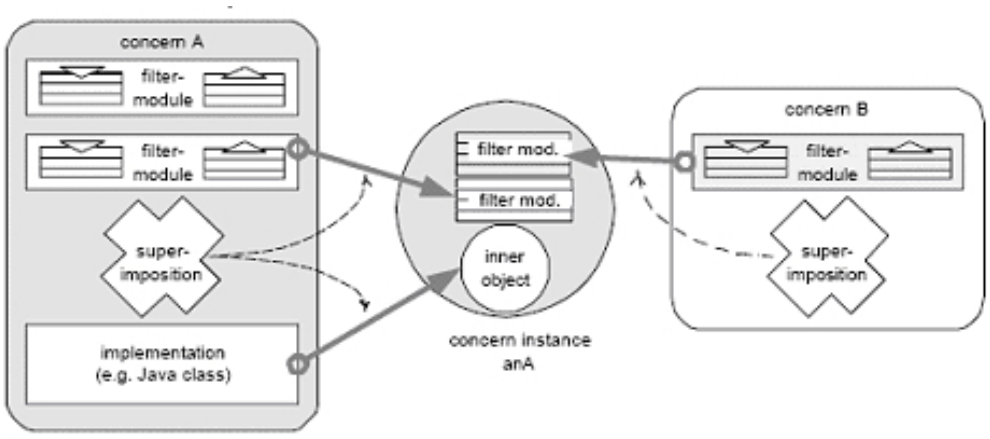


Figure 5.1: A Composition Filters Principle [11].

and return types of operations. [61]

Object-based is not the same as object-oriented, because object-oriented languages require inheritance support. An example of an object-based language is Visual Basic 6.0.

Programs are built up from a combination of program elements of the implementation language. This combination forms the static structure of the program and is used by composition filters to designate the join points in an application. [37]

Composition filters use program elements from the object-based paradigm that do not have an alternative in C, for instance objects. In this chapter we propose a mapping for composition filters to C. This mapping is discussed based on the key elements of composition filters and how these elements are selected or created with a composition filters concern. These elements are messages, superimposition and filter modules. Listing 5.1 presents a template with all the important elements of such a concern. This template shows that the concern is split up in a filter module and a superimposition part, which both contain several elements.

```
concern MyConcern ; {
    filtermodule MyFilterModule {
        internals
        externals
        conditions
        inputfilters
        outputfilters
    }
    superimposition {
        selectors
        filtermodules
        annotations
    }
}
```

Listing 5.1: `Composition Filters concern elements`

5.1 Message

A message is used as an interaction instrument between objects. Messages contain information about the target object, method, and possibly parameters. The sender object sends a message by calling a method in the target object, the target object then executes the intended method and returns control to the sender [50]. A method call is seen as a service request of an object, consistent to the intended behavior of the object. Composition filters intercept messages at the sender and/or target object's interface and possibly manipulate them.

A C program consists of functions and external variables, possibly divided over different translation units. The execution of a C program starts with the function *main*, followed

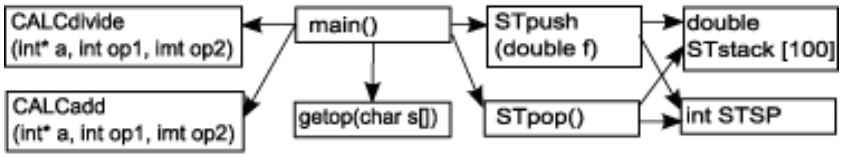


Figure 5.2: Messages in a C program. [42]

by a sequence of calls and executions of other functions. Information between functions is shared by calling a function with arguments or by manipulating external variables. [41]

The communication between functions is presented in Figure 5.2, where a C program is shown. The figure describes an adjusted version of the PolishCalculator from [42]. The program starts in the function *main*, and retrieves one by one operands with the *getop* function. Depending of the operand retrieved, the *STpush* and/or *STpop* functions are called and the right calculations are executed, *CALCdivide* or *CALCadd*. The *STpush* and *STpop* functions share the external variables *STstack*, where values are written and read, and *STSP*, which controls the size of *STstack*. Therefore the arrows in the figure can be seen as the messages that are sent in a C-program.

In essence, a message in C is very similar to a message in object-based languages. In both approaches functions or methods are called and executed. Object-based languages contain objects that encapsulate methods and variables. With a clear interface the state of the object is controlled. Also, in C functions update external variables, these variables are shared with all functions in the same scope.

So we can distinguish two principles of sending messages in C, updating an external variable and calling a function. However, variables are always updated with functions, just as in our example the functions *STpush* and *STpop*. Because of this we will only focus on function calls in our research.

5.2 Superimposition

Superimposition is developed as a technique to add filter modules to concerns. The use of superimposition is therefore associated with the concept of objects in object-oriented programming languages, but objects are not supported by C. In this section we discuss to what concern, or module, a filter module should be superimposed in C, and how we can select it with the superimposition part of the concern specification.

5.2.1 Module

Objects are a way to modularize programs. A module is illustrated by the following definition: "Every named program element that can be implemented as an independent entity can be seen as a module. A well designed module has a single purpose and a narrow interface to other modules. A module supports abstraction [61]."

In the concept of composition filters the most important characteristic of a module is that it sends and receives messages. Therefore the element that should serve as a module has to encapsulate one or more functions. In C the "largest" language unit which can abstract information is a *function* and therefore the most logical program element that could serve as a module. Because the function name is unique in a C-program, it is possible to weave at all the join points in the program. However, selecting a function as a module, limits the generality of selecting join points, because it is then only possible to superimpose a filter module at one function, Listing 5.4 or all functions, Listing 5.3, in the program.

```
1 superimposition {
2     selectors
3       oneFunction = { C | isModuleWithName(C, 'STpop') };
4     filtermodules
5       oneFunction <- fm;
6 }
```

Listing 5.2: `Superimposing one function`

```
1 superimposition {
2     selectors
3       allFunctions = { C | isModule(C) };
4     filtermodules
5       allFunctions <- fm;
6 }
```

Listing 5.3: `Superimposing all functions`

There are some techniques that can create a set of functions. Creating a set of functions offers a more extensive pointcut model, because of the possibility to weave at all the functions in the set. These techniques are:

File

A file is the only unit in C that encapsulates functions. A file defines the scope of the program elements in it. However, a file does not exist during computation, so it is not a program element, or value itself [61]. Because files are the input sources for the compiler, there is a possibility to retrieve the file name at compile-time. Also, the name at runtime can be retrieved with macros. A file is a convenient way to separate large programs in logical sets of functions. Therefore it is possible to use a file to select a set of functions, and in this way to select join points in a general manner.

Directory

A directory is not an element in C, however it is often used to logically group files and therefore functions. At compile-time it is possible to know in which directory the functions are found, because the file is used as input to the compiler, and it is possible to find out in which directory the file is located. The properties of a directory are very similar to a file, because it is not a program element. However, a directory may have subdirectories, which creates subsets and therefore more general pointcuts. Nevertheless, a directory may contain one or more files, so the number of functions it contains is at least equal to the number of functions in one file, but normally larger.

Naming Convention

In C it is usual to create logical units based on naming conventions. A naming convention is a pattern or series of patterns that prefixes an identifier of one or more elements in a program. If naming conventions are applied to program elements that exist during computation, then the naming convention is also available during computation. When naming conventions are applied to functions, a logical set of functions can be created that shares the same prefixed pattern. By prefixing a series of patterns it is possible to create subsets. A naming convention is an agreement, its fulfillment is not verified by for instance a syntax checker. Therefore it cannot be expected that naming conventions are applied without mistakes [45]. However, when naming conventions are applied with care, it can divide a program into unique sets of functions.

Annotation

An annotation is meta-information bounded to a program element that does not influence the execution process. Annotations can add multiple properties to program elements, and they are separated from the program elements, unlike naming conventions. Annotations can be used to create a set of functions, by adding an annotation with a specific property to the functions. It is possible to create sets that overlap and to create subsets, because an element can have multiple properties. This is very similar to naming conventions that need to be applied to the identifier of every function in the set. Nevertheless, annotations are not supported by C.

Table 5.1 summarizes the analysis of the techniques to group functions. We examined which kind of sets of functions can be created with the techniques: sets, subsets, and overlapping sets, because they have a positive effect to the generality of selecting join points. Also is the support of the techniques for C evaluated, because it is important that the techniques can be applied in existing projects, otherwise the tool is not able to re-engineer a program.

Table 5.1 shows that files can only create a set of functions, and not a sub- or overlapping set. The pointcut model of files is therefore the most limited of the techniques. A directory also supports subsets of functions, however normal use of directories result in large sets. Annotations can create the widest range of sets of functions, and there-

	Set	Subsets	Overlapping sets	Language support
File	+	-	-	+
Directory	+	+	-	+
Naming Convention	+	+	-	+
Annotation	+	+	+	-

Table 5.1: Supported properties function set techniques

fore offers the most extensive pointcut model. However, annotations are not supported by current C-compilers, so it is unwise to define modules with annotations. The most logical technique for creating a set of functions are naming conventions. Naming conventions support sets and subsets of functions and are commonly applied in projects. So in this case naming conventions are the best technique for creating sets of functions. In Listing 5.4 a filter module is superimposed to the *STpush* and *STpop* functions of Figure 5.2.

```
1 superimposition {
2     selectors
3       setFunctions = { C | isModuleWithName(C, 'ST') };
4     filtermodules
5       setFunctions <- fm;
6 }
```

Listing 5.4: `Superimposing a set of functions defined by a naming convention`

However, in C projects it is rather common to use a combination of the above techniques to define modules. For instance, in the paper of [4] naming conventions are used to create subsystems with one task based on file naming conventions. ASML uses naming conventions and directories to define modules from functions, where one file in the module serves as an interface to the other modules in the program. Because of this, we want to offer support for all the described techniques.

To apply composition filters to C, the concept of a module is not required. In C, function names are unique and because messages in C are functions calls, it is already possible to intercept all messages. However, for the generality of superimposing functions, selecting a set of functions is convenient. The most suitable technique for creating a set of functions is based on naming conventions. But there are more techniques that are used in existing systems. For composition filters we see a module in C as a set of functions defined by a file, directory, naming convention, annotation, or a combination of these techniques.

5.2.2 Superimposition of the concern specification

The three most important elements of the superimposition part of composition filters concerns are the selectors, filter modules, and annotations. These elements and what they mean in C will be explained here.

Selectors

The selector is a query over program elements that results in a set of modules. In the selector all possible program elements of a language can be used. A C program is a combination of external variables and functions, which may have parameters and local variables. These elements all have a type. It is also possible to select modules that contain elements with specific annotations. Because modules are a particular set of functions, it is possible to select a set of functions that all share the same program element or are combined by the techniques file, directory, annotation, naming convention, or a combination of these techniques.
However, selecting a program element based on a part of its name requires an extension to the selector with a general way to select names, so modules created with a naming convention can be selected as seen in Listing 5.5. Nevertheless, this is error sensitive. A better way to create a module is to extend the language model with a module element. In this way a module can be selected with a module query, Listing 5.6. Nevertheless a manner to parse naming conventions is needed.

```
1    selectors
2      moduleCALC = { C | isFunctionWithName(C, 'CALC*') };
```

Listing 5.5: `Selector with general way for name matching`

```
1    selectors
2      moduleCALC = { C | isModuleWithName(C, 'CALC') };
```

Listing 5.6: `Selector with module selection`

Filter modules

In the filter modules element, filter modules are bound to the selectors in order to superimpose them. With this element specific filter modules are added to the selected modules.

Annotations

An annotation is meta-information bounded to a program element which does not influence the execution process. Composition filters have the ability to select modules to superimpose on, based on program elements that have specific annotations, and to introduce annotations to a set of selected modules. Introducing annotations is done with this element, which makes it possible to bind annotations to selectors. Using the selector language for superimposing annotations to modules prevents the scattering of annotations, and reduces the inconsistencies of adding annotations manually [45].
Annotations are not supported by C, but they are a welcome extension because C lacks design information. At the moment C only supports comments and naming

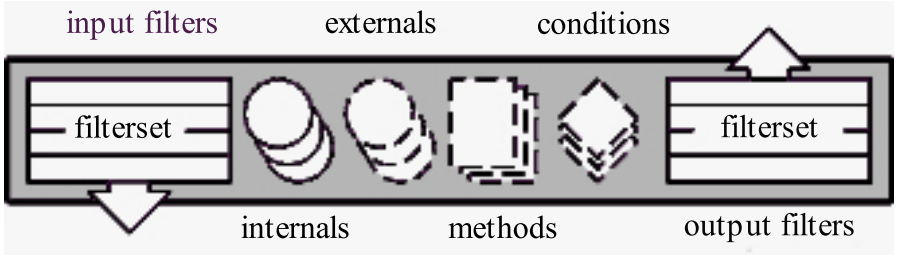


Figure 5.3: Filter Module

conventions as manners to add meta-data to a program. But comments are ignored by the compiler and therefore not usable to superimpose on. Naming conventions are agreements over information that has to be added to names of program elements. Program element identifiers are often used in existing programs to define a module, so it is unwise to use it for adding extra design information. As an example, in the Dynamic Execution Tracing concern of the crosscutting examples from ASML Section 3.2 design information is needed that describes if the parameters are written, read, or both in the function, in order to know which checks need to be performed. This information requires data-analysis of the code. By adding annotations this information is made available to the compiler.
Nevertheless, the selector query results in a set of modules. Therefore annotations can only be introduced into a module, and not for instance to add design information to parameters of functions. Also selecting a set of modules requires the manual addition of design information for defining modules. Therefore, we do not think that superimposing annotations to modules will address the problems, reusability, changeability and understandability of programming code in C. Because we do think annotations are a desirable extension to C, we will create support for annotations, but not for superimposing them.

5.3 Filter module

A filter module is similar to a set of filters. With a filter it is possible to intercept messages and manipulate them. The filter module and filter itself can best be explained by the elements it contains. Figure 5.3 shows a filter module with its elements.

5.3.1 Filter module of the concern specification

Internals

By declaring an internal element, state can be added to the superimposed module. An internal is fully encapsulated and therefore its state can only be used by one instance of a filter module. In practice using an internal means that intercepted

messages can be redirected toward functions of these elements, or functions of these elements can serve as conditions. Therefore these elements are a function or have the ability to encapsulate functions.
In C, functions are always declared external and therefore shared with other functions in the same scope, just as they share the external declared variables with these other functions. An internal element only has meaning as its state, used function plus the used external variables and functions, belongs to one filter module. This is only possible if only one instance of a filter module may be added to a scope, which is an unwanted constraint.
Furthermore, C does not support the instantiation of elements. Making internals work with C requires a unique identifier of the redirected function, the used external variables, and the used functions, for every instantiation of the filter module, because function and external variable identifiers always point to the same object. This is the only way to make sure that external variables and static local variables of used functions are not updated by other instances of the filter module. Making a copy of all the used elements and changing the right calls to the elements, in the advice which has to be woven as well as in the copied functions, is probably hard to implement.
Internals do not really fit the concept of C, which is that every element is declared external except for local variables, and making internals work with C will probably be very hard. For these reasons we do not offer support for internals in the the concept of composition filters in C.

Externals

Just as an internal, an external element adds state to the superimposed module, and therefore offers the possibility to redirect messages and use conditions from it. The difference between an internal and an external is that an internal is fully encapsulated, in contrast to an external that shares the state with all the instances. Unlike internals, externals do fit the concept of composition filters, because conditions and functions used from an external are only defined once in the program. So every filter module uses the same condition or function to redirect to.
Using conditions or functions of an external in a superimposed function, requires that they are declared in the same scope as the function. The used external variables and functions should be in the same scope as the definition of the function. This is possible by including a header file with the declarations of the used functions.
In Section 5.2 is presented that a set of functions can be defined by use of several techniques. These techniques can be used to create an external. By including the headers of the files this external covers, the state of the external is added to the superimposed module.

Conditions

Conditions need to be fulfilled in order to apply a filter. Conditions are therefore methods that return a boolean, which in C normally is implemented as a function that returns a 1 or not 1. Conditions may be declared in the external, in this case

the condition identifier will be preceded with the external name. A condition can also be a function that is local to the superimposition unit, so in the same scope.

Input filters and output filters

Composition filters support input and output filters. An input filter intercepts the incoming messages, whereas output filters intercept outgoing messages. This is a difference between a function execution and a function call. To intercept messages input filters need to be placed before the first statement of the function is executed and output filters before the actual call is done. It is important for the behavior of composition filters that the order of the filters at a join point stays preserved. As seen in Section 2.2 a filter contains a name, a type, and an ordered set of filter elements. A filter element has a condition, matching and substitution part. The matching and substitution part is used to intercept messages and redirect them. They both contain two elements: a target which is the intended object, and a selector which is the called method. In C a target is the set of functions that can be seen as a module. It is optional for use but not required, because a function name, the selector, is already unique and therefore the message is unique. When we want to intercept messages from more than one function, the target has to be used.

5.4 Example composition filters in C

In Listing 5.7 error handling is added to the *STpush* and *STpop* functions of the stack example, Figure 5.2.

The naming convention, "ST", creates a module that controls the stack and is used in the selector, line 16. The filter applied to module *ST* makes sure that the function *STpop* is not carried out when the *STstack* is empty, and no value will be pushed when the *STstack* is full.

```
1  concern ErrorConcern{
2    filtermodule error {
3      externals
4        error : ER;
5      conditions
6          emptyStack : error.ERemptyStack();
7          fullStack: error.ERfullStack();
8      inputfilters
9        error_filter : Dispatch = {
10             emptyStack=>[*.STpop] error.ERprintErrorMessage,
11             fullStack=>[*.STpush] error.ERprintErrorMessage
12       }
13   }
14   superimposition {
15     selectors
16       stackFunctions = { C | isModuleWithName(C, 'ST') };
17     filtermodules
```

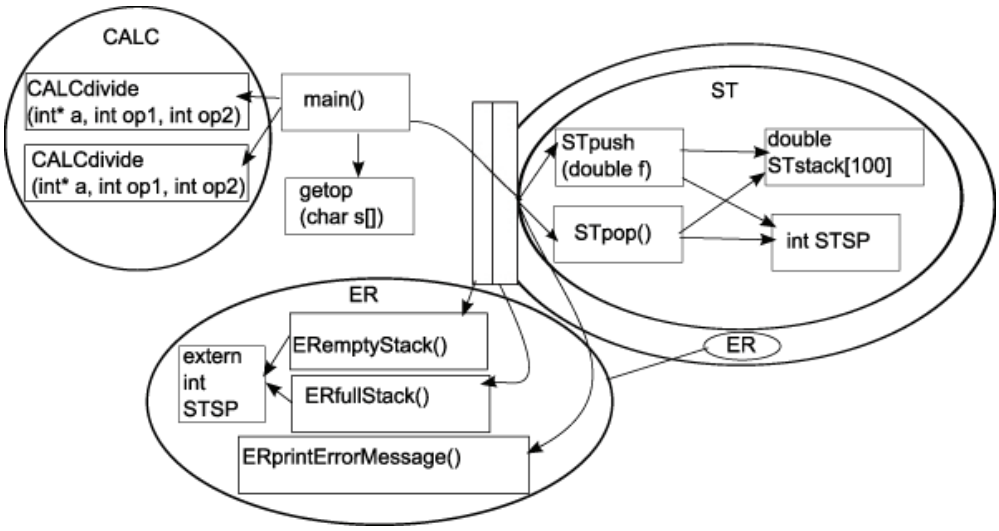


Figure 5.4: ErrorConcern applied to C program

```
18        stackFunctions <- error;
19    }
20  }
```

Listing 5.7: `Error Concern`

```
1  int ERemptyStack();
2  int ERfullStack();
3  int ERprintErrorMessage();
4  extern int SP;
5
6  int ERemptyStack(){ if(SP==0) return 1; return 0; }
7  int ERfullStack(){ if(SP==STACKSIZE) return 1; return 0; }
8  void ERprintErrorMessage(){printf(``Stack out of bounds exception'');}
```

Listing 5.8: `Error Concern implementation`

This Error concern is also applied to the program seen in Figure 5.2, the new program is seen in Figure 5.4. This is a conceptual figure that shows how composition filters can be applied to C. In the program the module *ST* is superimposed with the filter module *error*. This filter module intercepts messages to *STpop* and *STpush*. When a message to *STpop* is intercepted the condition *ERemptyStack* is evaluated, for *STpush* the condition *ERfullStack*. These functions are declared in the external *ER*. If one of these conditions is true, the filter dispatches to *ERprintErrorMessage*.

5.5 ASML concerns with composition filters

In section Section 3.2 four crosscutting concerns from a legacy system of ASML are explained. The code belonging to these crosscutting concerns is placed at the following join points:

- Before and after function execution;
- Insert variable declaration in function;
- Globally insert field declaration into a structure.

However, the Composition Filters approach is based on intercepting messages and manipulating them by use of filters, with a message defined as a function call. Compose* shall therefore not regenerate the original code, but it can create a program with the same functionality.

Composition filters supports input and output filters, therefore messages can be manipulated at the beginning of a function execution or before a function call. The way a message is manipulated is specified by the semantics of a filter. In Section 2.2 the filters that are supported by Compose* at the moment are described. It is possible to extend this set of filters in Compose*, by creating a custom filter.

The meta filter is very expressive and can be seen as around advice, which is common in the AspectJ approach. The meta filter increases the number of positions where Compose* can manipulate messages with after function execution and after function call.

To show how composition filters can implement the crosscutting concerns from ASML, implementations of the *Timing* and *Parameter Checking* concerns are given. The *Dynamic Execution Tracing* and *Function Naming* concern are not worked out. The *Timing* and *Parameter Checking* concerns cover the problems of the *Dynamic Execution Tracing* concern, therefore this shall not show anything new. The *Function Naming* concern is only used by the other concerns, and because the function name is available in the message this concern is not needed anymore.

5.5.1 Timing concern

The Timing concern measures the execution time of a function. It starts timing at the beginning of a function with a call of function *TIMING_in* and stops at the end of the function with a call of function *TIMING_out*. These functions use a global and local *TIMING_handle*. These four pieces of code characterize this concern in the system.

Defining a global or local does not fit the concept of composition filters. Therefore another manner to weave this piece of code into the base code needs to be found, which preserves the functionality of the system.

For usability reasons the external variables of the *CALC* module are all grouped in one element, a struct. These variables are shared between all the functions in *CALC*, just as the global TIMING_handle variable. When the Timing concern is separated from *CALC*, the global needs to be defined in the Timing concern, and is not needed anymore in *CALC*.

The local *TIMING_handle* variable is specific for a *TIMING_in* and *TIMING_out* call in a function. Therefore this variable should be defined in the same scope as the functions. The *TIMING_in* and *TIMING_out* functions are called at the beginning and end of a function. Because the functions, the variable and the arguments of the functions are for every function the same, it is possible to create a custom filter that adds the three pieces of code to every function.

In Listing 5.9 the implementation of the Timing concern is seen. The custom filter weaves the code at every function found in module *CALC*, line 6 and 10. The global *ti_handle* is externally declared in Listing 5.10 and therefore shared with the superimposed module *CALC*.

```
1  concern Timing{
2    filtermodule timing {
3      externals
4        time : TIMING;
5      inputfilters
6        timing_filter : timing = { [*.*] }
7    }
8    superimposition {
9      selectors
10       calcFunctions = { C | isModuleWithName(C, 'CALC') };
11     filtermodules
12       calcFunctions <- timing;
13   }
14 }
```

Listing 5.9: `Timing concern` with composition filters

```
1  TIMING_handle ti_handle;
```

Listing 5.10: `Timing concern implementation`

5.5.2 Parameter Checking

The implementation of the *Parameter Checking* concern in composition filters is seen in Listing 5.11. It is originally implemented as an if-statement where a parameter is checked if it is *NULL*. If this is the case an error function is called and the function returns with an error value. This is done for all the in-, out-, inout- parameters.

This is similar as the functionality of a dispatch filter, which redirects a message to another function, if a condition is true. However, a condition is a function declared in an

external. This condition does not know the parameter values, and if they are annotated as in, inout, or out. When a dispatch filter is used, every superimposed function, with different parameters, should have its own condition. This reduces the general applicability of the concern and therefore the effectiveness of using aspect-oriented programming.

At compile time the annotation information, parameter names and function name are available. So it should be possible to create a custom filter that checks the parameters values, calls the right error message and returns.

```
concern ParamCheck{
  filtermodule paramcheck {
   inputfilters
      paramcheck_filter: Paramcheck= { [*.*] }
  }
  superimposition {
    selectors
      calcFunctions = { C | isModuleWithName(C, 'CALC') };
    filtermodules
      calcFunctions <- ParamCheck;
  }
}
```

Listing 5.11: `Tracing concern` with composition filters

5.6 Summary

In this chapter the concept of composition filters in C is worked out. The Composition Filters approach is developed for object-based programming languages, and uses concepts from these languages that do not exist in C.

There are three important elements in composition filters: messages, filters, and superimposition. Messages in composition filters are method calls, which are very similar to function calls in C. Filters can intercept messages and manipulate them, an input filter can do this before function execution, an output filter before function call. With superimposition a set of functions can be selected, where a filter module can be put on. The set of functions can be defined by a file, directory, naming convention, or annotation.

A composition filters concern is constructed from elements. A mapping of these elements to C is made in this chapter. Externals can be used to add state to a concern. It is a module, however to prevent missing global variables, in practice the header files these modules overlap are added to the concern. Conditions are functions that return a 1 or not 1 and are defined in the external or in the scope of the superimposed functions. With the selector language it is possible to select a set of modules, based on its name, global or function it encapsulates, or a parameter or local variable of a function. Annotations are a useful extension to C, because extra information can be added to program elements without performance loss. This information can be used to superimpose on.

Our proposed design of composition filters in C is applied to an example, and the cross-cutting concerns of ASML. It was possible to transform both into an aspect-oriented design.

Chapter 6

Weaving technology

Weaving is the process of composing core functionality modules with aspects, thereby yielding a complete system [32].

The concept of composition filters is language independent, just as a large part of Compose⋆. One of the functions of Compose⋆ which is platform specific is the weaving process. Compose⋆.NET uses a runtime weaver that changes Intermediate Language(IL)-code. This implementation was chosen, because the most important requirement at the time was: flexibility. This weaver can be used by all .NET languages, because they all use IL, and it can change the behavior of the advice during execution. Flexibility is also the most important requirement for the Compose⋆.Java implementation, therefore it also supports a runtime weaver.

For Compose⋆ C other requirements apply. Some requirements are already explained in Section 3.4. In this chapter the set of requirements shall be extended or adapted to make the concept of composition filters work in C, which is described in Chapter 5.

These requirements are used to find a suitable aspect-oriented weaver for C that can handle the weaving process. We favor an existing weaver instead of developing our own, because of the limited time available for this project.

6.1 Requirements for the weaver

In the previous chapter we proposed the concept of composition filters in C. To implement this, weaving technology is needed that can realize this concept. In this section functional requirements are derived from the proposed concept of composition filters in C. The the quality requirements, which are explained in the problem identification chapter, are also analyzed and adapted, if needed.

Functional requirements

In Compose⋆ C a message is a function call. For intercepting messages, Compose⋆ C needs to have the ability to superimpose filters at one function or a named set of functions. Filters are divided in input and output filters, with the difference that input filters intercept messages at the execution of the function and output filters intercept messages when a function is called. An input filter should therefore be placed before the first statement of a function is executed and an output filter before a function call. A meta filter is an exception to this, because it is defined as around advice, so it can also place advice after a function call or execution. This means that the weaver needs to support a pointcut that can select one or more functions based on a name, an execution and call join point, and before and after or around advice.

Using externals in C requires that a function, or set of functions, and their used external objects, need to be introduced into the same translation unit as the superimposed function. It is possible to achieve this, by weaving an #include statement with the file where the function and objects are declared, or by declaring all the external objects extern in this file. The latter has the advantage that the preprocessor does not have to be called again, but it is probably harder to realize. So the weaver should support a way to add global declarations or an #include statement to a translation unit.

The weaver does not have to support a manner to weave advice in a generic manner, because the compile-time of Compose⋆ offers all the superimposed functions and the filter modules belonging to them. In a composition filters concern it is possible to use a general manner for selecting functions. If naming conventions are applied in a project Compose⋆ makes it possible to use them.

Filter-sets are a sequential composition of filters. In object-oriented languages every incoming message will travel through the composition of the input filters, and outgoing messages through the composition of output filters. In C filters are woven inside the functions and therefore only messages that match the matching part of a filter are traveled. It is possible that there are filters with the same matching part that need to be in a specific order. Therefore, it is important that the weaver supports ordering at join points. In Listing 3.1 is seen that there is a dependency between the concerns. The func_name variable that is set by the Function Naming concern, is used by all the other crosscutting concerns. Because of this, the concern has to be woven before the others. This is another reason why ordering at a join point is needed.

In C there are no alternatives for annotations. Annotations are a welcome addition to C, so it would be convenient, if the weaver could support this in some kind of way.

The functional requirements are:

1. Pointcut that selects a function;
2. Execution join point;

3. Call join point;
4. Before and after, or around advice;
5. Possibility to introduce defined objects;
6. Manner to specify ordering of advice;
7. Interface to implement the aspects;
8. Annotations.

Quality requirements

The intention to develop the weaver is originated from problems with existing systems. C is still often used because of its efficiency. The effect of the interference of the weaver to the efficiency of the program should be as little as possible. It is not uncommon that the existing systems are big, such as in the case of ASML. Increasing compilation-time is therefore not desirable, so we prefer the prototype to be fast. It also is preferable to keep using the same C compiler. Therefore the weaver cannot be dependent of new language features, for instance variable length arrays that are supported by C99, but not by C89 and of course the new libraries. Restrictions to the working of the weaver by naming conventions or programming style are also not desirable. Furthermore, it should be possible to manage the weaver with Compose⋆. This means that it must be possible to generate the aspect with Compose⋆ and to automatically invoke the weaver.

The quality requirements are:

9. Implemented tool;
10. Fast weaving process;
11. Wide acceptance for existing C code;
12. Delivers an efficient program.

Discussion

For the development of Compose⋆ C some requirements are more important than others. An implemented tool is needed to use the weaver in Compose⋆ C. If there is no weaver, requirements 10, 11, and 12 are also automatically not available. Requirements 10, 11, 12, are desirable, just as the support for annotations, requirement 8, but they are not obliged for creating Compose⋆ C. However, we prefer support for all these requirements.

6.2 Aspect-oriented weavers for C

In the preliminary section, twelve requirements are proposed that should be fulfilled by the aspect-oriented weaver, in order to be suitable for integrating in Compose⋆ C. In this section we examine if an existing aspect-oriented weaver for C satisfies these requirements.

Most of the current weavers are briefly described in [1], where a similar research is performed. However, the requirements of that project do not cover ours. The weavers are split up in source code, and intermediate language weavers, because there are specific characteristics attributable to it. These approaches of weaving aspects are explained in Section 1.3.2. This section also shows that some weavers adapt the virtual machine, but we did not find such a weaver for C. The aspect-oriented weavers for C that we know of and we will discuss in this chapter are:

Source code	**Intermediate language**
AspectC	uDiner
AspectC(06)	TinyC2
AspectC++	Arachne
Aspicere	TOSKANA
C4	TOSKANA-VM
WeaveC	

Table 6.1: C Weavers

We discuss these languages one by one and judge them by the fulfillment of the requirements. However, this is not so easy. Aspect-oriented software development for C is a relatively new research. To evaluate the requirements, information of some weavers had to be extracted from papers, because the weavers do not exist yet, or are not ready for use. Some requirements are probably not yet considered by the developers of a few weavers, because they are not mentioned anywhere. Because it is likely that there is no support for the unmentioned requirements, we negatively judge them.

6.2.1 Source code weavers

Source code weavers combine the base code of the program with aspect code. Source code weavers generate regular C code, which is interpretable with a normal C compiler. This approach of weaving has the advantages that, high-level sources are modified, optimization of the code by the C compiler is still supported, and it is possible to use different C compilers. Nevertheless, source code weaving also has some drawbacks, a source code weaver is language dependent, and its expressiveness is limited to the expressive power of the source language. A more extensive explanation of these advantages and drawbacks can be found in Section 1.3.2.

AspectC

AspectC is based on the AspectJ approach. AspectC supports a function join point, call, execution, cflow and within pointcuts, and before, after and around advice [21]. Aspects in AspectC can be declared in a separate file, and are singletons [2]. This means that the requirements 1 till 5 and 7 are all supported by AspectC.

AspectC does not provide a general way to select functions to superimpose on. The precise name of the function has to be used in every pointcut. It also does not support information about the joinpoint that can be used in the advice.[1]

Also, AspectC does not offer support for annotations or other manners for adding or using meta-information, and no option to define an order is found. So requirement 6 and 8 are not fulfilled by AspectC.

Furthermore, there is no tool available of AspectC. It seems like the AspectC project stopped in 2003. This project never resulted in a release, therefore requirements 9 till 12 are not available.

AspectC

July this year, a new aspect-oriented weaver for C is released. This new weaver also caries the name AspectC [8], to avoid misunderstanding, in this thesis we shall refer to the new version with AspectC(06). AspectC(06) is based on the original AspectC ideas, but it is created by the developers of TinyC2. Because this weaver was not available at the time we selected a weaver for integration with Compose⋆ C, this weaver is not used. However, we do want this research to be complete, and that it will show a clear overview of the current weavers, so we added AspectC(06) to this chapter.

As we said, AspectC(06) is an implementation of the original AspectC idea. This idea included a function join point, call, execution, cflow and within pointcuts, before, after and around advice, and aspects as separate files. These features are all implemented in the latest release. Furthermore, AspectC(06) supports additional features, such as an infunc and infile pointcut, and it also has a general manner to select join points based on their name. So we can conclude that requirements 1 till 4, and 7 are fulfilled. However, the current version AspectC(06) does not support a way to introduce global objects, or to include header files, therefore requirement 5 is not supported.

Nevertheless, this AspectC(06) version does not offer support for annotations or other manners for adding or using meta-information, and there is no option to define a weaving order supported. So the requirements 6 and 8 are not fulfilled by AspectC(06).

Just as WeaveC, AspectC(06) creates an abstract syntax tree from the preprocessed source code. In this tree the advice is woven. After this, C-code is generated from the trees. The generated code is still efficient and the compile process is fast. AspectC(06)

can compile sources that are pre-processed by many different C compilers. This means that AspectC fulfills requirements 9 till 12.

AspectC++

AspectC++ is an aspect-oriented extension to C++. It is conceptually very similar to AspectJ. An aspect is declared in a separate file and is woven into the base code as a class. Because a C++ compiler can also compile C code, it is possible to build a working program. The intention of the developers is to create support for C. AspectC++ supports requirements 7,9,10, and 12, but it does not fulfill number 11.

AspectC++ is very extensive. It supports the pointcuts call, execution, cflow, base, derived and within. Also the pointcut-functions that, target, result, and args are available, these functions use runtime join point information. An ordering of pointcuts can be defined with a special advice, order. A join point can refer to a function, an attribute, a type (struct or union), or a point from which a join point is accessed. AspectC++ supports before, after and around advice and has the ability to introduce new elements in structs or unions. General weaving is supported by AspectC++, by use of match expressions to create pointcut expressions. These expressions can match type names as well as attribute and method signatures. A wildcard character is supported that matches any string. Nevertheless, AspectC++ does not offer a possibility for using annotations. So it fulfills requirements 1 till 6, however requirement 8 is not fulfilled. [56] [7]

Aspicere

An aspect in Aspicere is a C compilation unit extended with the possibility to contain advice. Aspicere defines pointcuts with prolog queries, which is a very expressive mechanism. It is possible to create generic advice, change the parameters of a superimposed function and it supports “weave-time metadata” [9], so it supports requirement 8.

Advice can be executed around a join point, nevertheless Aspicere only supports just one, being the function call. It does offer join point information by use of a thisJoinPoint-struct. The declared ordering of aspects and advice is also the order it is executed [1]. This means that it fulfills requirements 1, and 3 till 7, but not 2.

Aspicere even works on non-ANSI C code, nevertheless applying Aspicere to a test case showed that compilation is very slow [1]. Therefore it fulfills requirements 9,11, and 12, nevertheless 11 is not fulfilled.

C4(CrossCutting C Code)

C4 is a toolkit containing three source-to-source transformation tools, being a C4 weaver, C4 unweaver and a C4-to-C compiler. With C4 it is possible to define aspects in the

source code. With the unweaver the sources are split up into C4 aspects and base code. With the weaver and compiler the source code can be re-generated. Nevertheless, C4's unweaver and weaver are still under construction [20]. So requirements 9 till 12 are not available.

The intermediate C4 aspects are very similar as AspectC aspects. Advice can be invoked before, after, and around a join point. It supports the pointcuts, function execution, introduction of elements in structs and unions, and introduction of global variables. A call pointcut is however not supported. C4 does not support any join point information, a generic manner to weave advice, or possible support for annotations, just as AspectC [33]. So C4 does fulfill requirements 1,2,4, and 5.

When aspects are annotated in the source code, the problem of ordering is not a problem, however automatically annotating source code Compose⋆ C is not possible and therefore not usable. So we do not focus on annotating the source. Therefore we state that it is not possible to specify ordering, because the aspect language does not have an element for it. Requirement 7 is fulfilled by C4, but 6 and 8 are not.

WeaveC

Aspects in WeaveC can be defined with XML. These aspects make a clear distinction between pointcuts and advice. In this way multiple pieces of advice can be woven at one pointcut. A priority can be added to advice to guarantee that the advice is executed in the right order. WeaveC also offers support for annotations. This means that WeaveC supports requirements 6, 7, and 8.

WeaveC supports the pointcuts function call and execution, global and local field introduction, and element introduction in structures. Advice can be placed before or after a join point. It is possible to weave advice in a general manner. So the requirements 1 till 5 are all fulfilled.

WeaveC creates an abstract syntax tree from the preprocessed C code. Advice is added by transforming the tree, which is a very fast process. Therefore, WeaveC also fulfills requirements 9 till 12.

6.2.2 Intermediate language weavers

Intermediate language weavers can add behavior to a program, by directly weaving aspects in a target language. This target language can for instance be the Common Intermediate Language, or machine code. Weaving in an intermediate language offers more control over the executable program. The main advantages that intermediate language weavers have over source code weavers are, that they are, often programming language independent, more expressive, and source code independent, and also they can add aspects on load-, or runtime. However, intermediate language weavers also

have some drawbacks over source code weavers, the intermediate language is harder to understand, and it is more error-prone. For a more extensive explanation of these advantages and drawbacks we refer to Section 1.3.2.

μDiner

μDiner compiles code into shared libraries, which are loaded at runtime in the advised base application. With μDiner it is possible to weave around functions and global variables, as they are annotated as hookable. Ordering should not be a problem in this way. μDiner also supports a cflow-like pointcut that examines the nested call hierarchy [51]. μDiner is superseded by Arachne [1], therefore no tool is available. μDiner fulfilles the requirements 1, and 3 till 6, but it does not fulfill requirements 2, 7, and 8, and because it does not exist we state that 9 till 12 are not available.

TinyC2

TinyC2 uses a Dyninst library to enable runtime weaving. It does not work with aspects, but a similar construct that describes a join point and advice, called a "snippet". A snippet can contain the pointcuts onentry and onexit of a function. It is possible to select a function with its name, or to select all functions in the program with a wildcard character. In the snippets globals, argument and return values can be used. There is no option for creating an ordering [63]. Therefore TinyC2 supports requirements 1, and 3 till 6, but not requirements 2 and 7.

The TinyC2 snippets are translated to C++ code that invokes the Dyninst library. With a C++ compiler a binary executable is generated, which is linked to the Dyninst instrumentation library. TinyC2 is very slow because of DynInst [63]. Also TinyC2 is not freely available [1]. Because of these reasons we state that requirement 8 is not fulfilled and 9 till 12 are not available.

Arachne

Arachne is an improvement of the μDiner framework. Annotating the source code for specifying aspects is not necessary in Arachne. At weaving-time Arachne executes a specific rewriting strategy depending on a pointcut. For instance a function call, shall be rewritten with a jmp instruction to a hook. Depending on the evaluation of the hook, the hook executes the advice code, which is a function in Arachne, or calls the original function. [25]

Arachne supports the pointcuts function call, read and write of global variables, and cflow. Nevertheless, no execution join point is available and no join point information can be used, only arguments can be passed [25]. Specifying an ordering seems impossible,

and also annotations are not supported. So Arachne offers support for requirements 1, and 3 till 6, but not for requirements 2,7, and 8.

Arachne uses code splicing to call the hooks at runtime. Code splicing is fast, but it only works on computers that have an IA32 processor and a Linux kernel that supports anonymous memory mapping [6]. This means that requirements 9, 10, and 12 are fulfilled by Arachne and 11 is not.

TOSKANA(Toolkit for Operating System Kernel Aspects)

TOSKANA is a dynamic weaver that uses code splicing, just as Arachne, to jump to the address of the advice, which results in an execution of the advice. An aspect in TOSKANA is a C function with the returntype "ASPECT". TOSKANA can add advice before, after, or around in-kernel functions, at the execution pointcut. Selecting a pointcut is done with an "aspect_init" function, that combines the pointcut and the advice. TOSKANA therefore fulfills requirements 1, 3, 4, and 6, nevertheless 2,5,7, and 8 are not fulfilled. TOSKANA does not work with older Unix-like systems, because applications that are not executing in kernel context, do not have access to kernel memory space, which is a requirement for using TOSKANA [30]. There is no implementation known of TOSKANA, so requirements 9-12 are not available.

TOSKANA-VM(Toolkit for Operating System Kernel Aspects-Virtual Machine)

The inventors of TOSKANA also thought of an implementation using a low level virtual machine[31]. The intermediate code of this virtual machine is of higher level than binary code, which offers a possibility for more advanced join points, such as call, execution, variable assignment, and access. Also, no release of TOSKANA-VM is found. TOSKANA-VM offers support for the requirements 1 till 5, and 7, does not fulfill requirements 6 and 8, and because no tool is available, requirements 9 till 12 are also not available.

Overview

Table 6.2 briefly shows which requirements are fulfilled and which are not by the weavers.

6.3 Selecting a weaver

In the last two sections requirements for a weaver are described and existing weavers are analyzed. In this section we select a weaver that shall be integrated with Compose⋆ C.

	Functional requirements								Quality requirements			
Requirements	1.Function pointcut	2.Execution join point	3.Call jp	4.B+a/around advice	5.Introducing objects	6.Ordering	7.Aspect interface	8.Annotations	9.Tool	10.Fast weaving	11.Existing C code	12.Efficient program
AspectC	+	+	+	+	-	-	+	-	NA	NA	NA	NA
AspectC(06)	+	+	+	+	-	-	+	-	+	+	+	+
AspectC++	+	+	+	+	+	+	+	-	+	+	-	+
Aspicere	+	-	+	+	+	+	+	+	+	-	+	+
C4	+	+	-	+	+	-	+	-	NA	NA	NA	NA
WeaveC	+	+	+	+	+	+	+	+	+	+	+	+
μDiner	+	-	+	+	+	+	-	-	NA	NA	NA	NA
TinyC2	+	-	+	+	+	-	-	-	NA	NA	NA	NA
Arachne	+	-	+	+	+	-	+	-	+	+	-	+
TOSKANA	+	-	+	+	-	-	+	-	NA	NA	NA	NA
TOSKANA-VM	+	+	+	+	+	-	+	-	NA	NA	NA	NA

Table 6.2: Fulfilled requirements by the weavers

Table 6.2 shows that there are only five weavers that are implemented and available, being: AspectC(06),AspectC++, Arachne, Aspicere, and WeaveC. Nevertheless, at the moment the weaver was selected, AspectC(06) did not exist, so it was not an option for use and therefore not discussed in this section.

AspectC++ is probably the most extensive aspect-oriented language for C. However it generates C++ code. It should be possible to compile C code with a C++ compiler, however there are many incompatibilities. For instance C90 does not support "//" comments, and both C90 and C99 do not support booleans, C++ does. To apply AspectC++ to a C-system, it is likely that adjustments to the source code are needed. This is not desirable and therefore AspectC++ is not an option for use in Compose* C. AspectC++ also does not support any annotation manners. This was not really a requirement, but more a preference.

Arachne is a runtime weaver. A runtime weaver is normally more flexible, but slower than a compile-time weaver. However Arachne is not slow, because it uses code splicing. Unfortunately using code splicing makes Arachne dependable of a specific processor architecture, IA32, and a Linux kernel that supports anonymous memory mapping. This is unacceptable for Compose* C, because we want to make it general applicable. Furthermore, Arachne does not support a execution join point. An execution join point

is needed for creating input filters, an important element of composition filters. Also Arachne does not supports any form of annotations. Because of these reasons Arachne is not suitable for use in Compose⋆ C.

Aspicere only comes short at two requirements. It does not support an execution join point, needed for input filters, and the compilation process is very time consuming. Building a case program with Aspicere required 17 hours and 38 minutes, instead of the original 15 minutes without Aspicere [3]. No support for input filters and a very slow building process, is not preferable. Therefore Aspicere is not the right weaver for Compose⋆ C.

The last weaver is WeaveC. WeaveC supports all formulated requirements and therefore selected for use in Compose⋆ C. WeaveC supports the right join points, an option for annotations, and a manner to prioritize advice. It uses GCC to preprocess the C-code, and only delays the compilation process 1,5 times. The generated code can be compiled with GCC and therefore compiler optimization is still applied, so the program's efficiency is preserved. Nevertheless, the included GCC compiler is an older version, which does not support all the current options of C.

6.4 Summary

The weaving process of Compose⋆ is platform dependent. Adapting the current implementation is not an option. Developing the weaving technology our own is time consuming, therefore the current aspect-oriented weavers for C are evaluated. This evaluation is based on a set of requirements derived from the concept of composition filters in C, and some characteristics that Compose⋆ C should fulfill.

We examined eleven weavers, unfortunately only four were ready for use at the time, being: AspectC++, Arachne, Aspicere, and WeaveC. AspectC++ and Arachne are not general applicable in the C environment and therefore not suited to apply in Compose⋆ C. Aspicere lacks an execution join point, so input filters cannot be implemented, and its building process is very slow. WeaveC fulfills all the requirements, therefore WeaveC shall be integrated with Compose⋆ C in order to realize the weaving.

Chapter 7

Compose⋆ C

Compose⋆ is build from several components. The components can be divided in two groups, platform independent and dependent. Platform independent modules do not need to be adapted if a new language will be integrated. However, the platform dependent modules do need to be adapted or replaced. In Figure 2.3 the architecture of Compose⋆.NET is shown. This figure shows that only the compile-time is language independent.

In this chapter we explain the implementation and design decisions of the language dependent components for C. The main functions that are adapted, created, or extended are: creating a C language model, the collection of program-information, adding a source weaver, implementing the filters, and adding an IDE-plugin. These functions are discussed in the next sections.

7.1 Language Model

Selecting static join points in a program requires information about the static structure of the program. The static structure of a program is a combination of program elements belonging to the implementation language. With the selector of a composition filters concern it is possible to query the elements in order to select the right join point. [37]

In the architecture of Compose⋆, Section 2.4, is seen that the design of Compose⋆ is build around a repository. This repository contains information about the program created with Compose⋆. It is possible to extend the repository with a language model of the implementation language. The language independent part of Compose⋆ contains a meta-model that already defines program elements seen in most programming languages, as well as their properties and their relations. The elements in this model can be extended to create a language model for the programming language to which we want to port Compose⋆, in this case: C.

This section presents the proposed language model for C, explains how this model is implemented in Compose⋆, what problems it caused, and how these problems are addressed.

7.1.1 C Language Model

In Chapter 4 most of the program elements of C are already explained. The chapter concludes that C has a limited type-system consisting of only a few basic-types, and the possibility to create derived types, being: function, pointer, array, struct, union, or enum. Running a C-program is an execution of a series of functions that call each other. They can use external, or local variables, or parameters, but always have a return-type.

Chapter 5 explained the concept of composition filters in C. This chapter discusses that C does not support a program element that groups functions, but that there are some techniques that can be used to create a set. The availability of such a set enhances the generality of selecting join points, which is a function call or execution. Sets of functions can be created by using one of the following techniques:

- Files;
- Directories;
- Naming conventions;
- Annotations.

Furthermore, we saw that annotations are not only useful for defining a set of functions, but also to add extra design information to programs. It would be convenient to add annotations to most program elements.

The language model for C, which supports all the elements, properties and relations, and makes all the features that are discussed in Chapter 5 possible, is seen in Figure 7.1.

As we can see in the figure there are some inconsistencies in the model which probably need some explanation. These inconsistencies are:

ProgramElement
: Only the elements available during execution are program elements, being variable, parameter, type, and function. The other elements are derivable from the file-system, or identifier names. Annotations are available at compile-time, but not during execution and therefore not defined as a program element.

Type
: In the language model no distinction is made between basic-types and derived-types. By generalizing a struct, array, pointer, enum, and union as special kind of type, it is possible to derive more structure information of the program. In Compose⋆ C we only want to query the program based on the return-type or type of parameter of a function, or the type of an external in the program. We do

not want to query how the derived-types are constructed. Therefore, basic and derived-types are not split up, and both can be queried based on their name;

Variable
Variables can be declared local or external, which results in a different parent. Because, other properties match they are defined as one program element.

Annotations
Are only applied to the elements that exist during computation, the actual program elements. Annotations can not be added to types, but they do have a type;

Subsets
With directories, naming conventions, and annotations it is possible to create subsets of functions. With directories and naming conventions this is done by creating a parent-child structure of the same elements. With annotations this can be done by applying multiple attributes.

Overlapping sets
With annotations it is also possible to create sets of functions that overlap, again by using multiple attributes.

Sets of multiple techniques
Creating a set of functions can be done by combining techniques. It is possible to create a set of functions, for instance based on files in directories, or by combining naming conventions and files.

7.1.2 Abstract language model

The language independent part of Compose* holds an abstract language model. This model is extendable to other programming languages. This model is seen in Figure 7.2.

This model supports program elements which are not supported by C, being: class, interface and namespace. The elements field and method are object-oriented, but very similar to the variable and function elements in C. In the abstract language model is seen that a type has the same connections to a method, parameter, and variable, but it is extended with the elements class and interface, which are not supported in C. As we see from this model, it covers elements of most object-oriented languages, because it supports classes and objects that are first class values.

Extending this model to create the model of Figure 7.1 is impossible. For instance the element file in C should extend the element type in the abstract language model, because this is the only element that contains functions and variables. However, this makes it possible to have a file as a return-type of a function, which is not possible in C.

Nevertheless, the working of Compose* is dependent of the implementation of this abstract language model. It is mandatory that the elements type and method are derived from the program.

Type
In Compose* it is only possible to superimpose on elements which are an instance

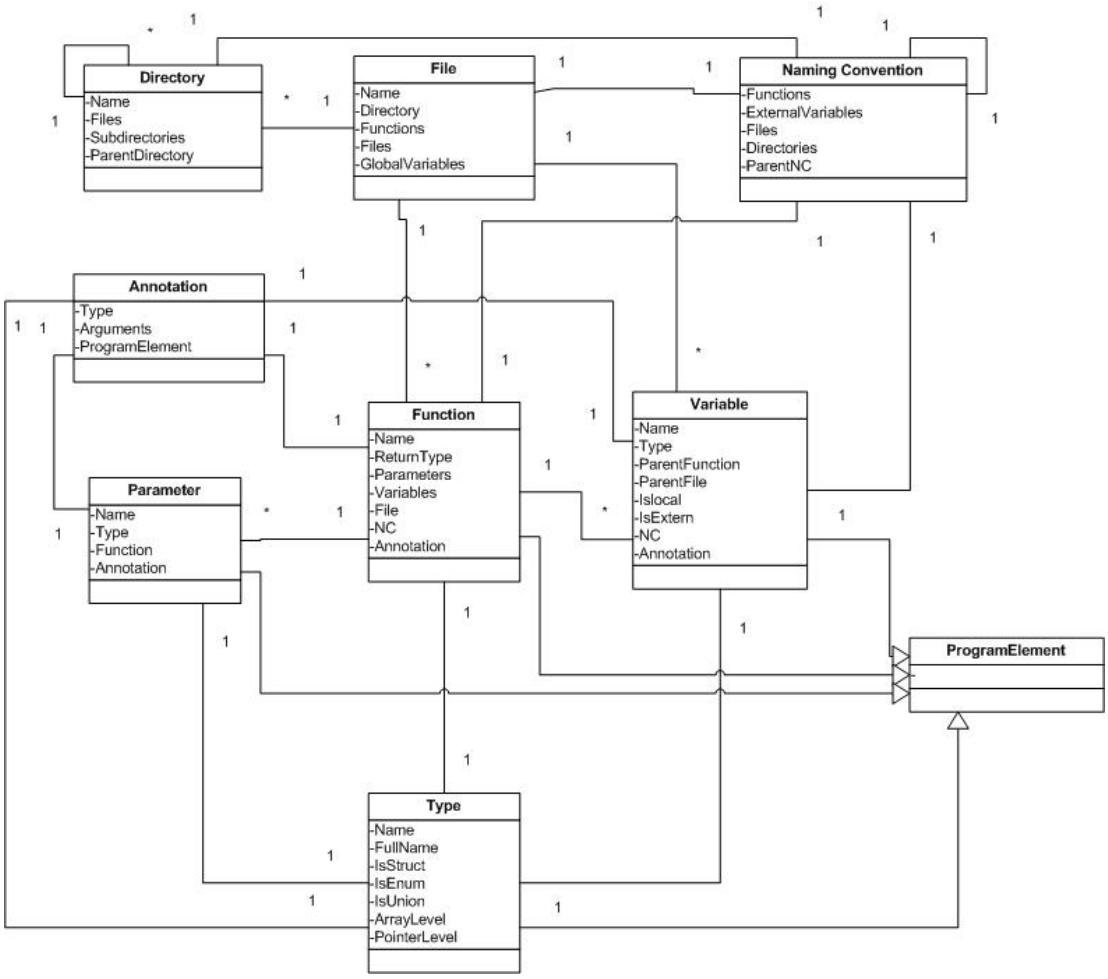


Figure 7.1: C language model

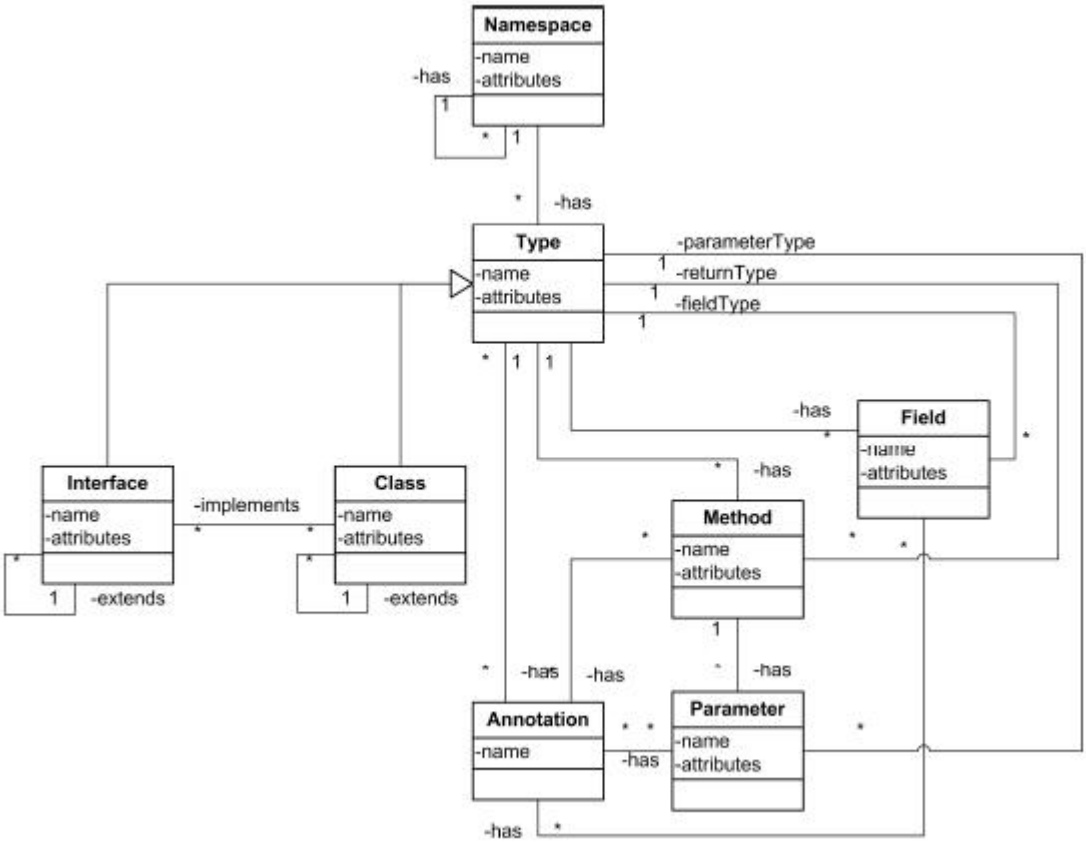


Figure 7.2: Abstract language model

of type.

Method

A method is the element used in the filters of the Compose⋆ concern. Because the signature of the method is evaluated, also the return-type of the method has to be a type and the parameters need to be an instance of parameter.

7.1.3 C language model in Compose⋆ C

For adding the program structure information to the repository, we considered not extending the abstract language model, because it was impossible to fit the C language model to this model. This was however not an option, because the relationships between the abstract language model elements that are used by the selector element, need to be defined in the extensions of these elements. Because of this reason we have implemented a C language model.

The implemented C language model is very straight forward:

- File is an extension of Type;
- Function is an extension of Method;
- Parameter is an extension of Parameter;
- Directory is an extension of Namespace;
- Type is an extension of Type;
- Variable is an extension of Field;
- Annotation is an extension of Annotation.

The relations between the extended elements are the same as shown in Figure 7.2.

One of our requirements for Compose⋆ C was that the composition filters concerns should be compatible with the other Compose⋆ implementations. In the selector element of other implementations, object-oriented elements are used which do not have a meaning in C, being: class, method, and namespace. Because the implemented language model is an extension of the object-oriented abstract language model there is a clear mapping possible. This mapping is specified in a prolog query definition file that comes with the Eclipse-plugin.

In the implemented C language model there is no place for the Naming Convention element of Figure 7.1. A naming convention is a specific set of characters that prefixes the identifier of elements. If we want to create a concern based on naming conventions, it would be possible to add such an element and let it extend the element Type. In this way it is a set of functions and it is possible to superimpose Naming Conventions. However, the patterns of the naming conventions should be specified somewhere, and every time a function is added to the language model, it should be evaluated to which naming convention it belongs. This is a laborious manner for adding naming conventions to Compose⋆ C. There is a much easier way. The selector language makes use of a prolog engine, which supports a query that checks strings for character patterns: *matchingpattern(string, pattern)*. By using this query it is possible to select files that contain functions with a specific query in the composition filters concern self. Using this query does not require any alterations to Compose⋆ C and therefore is chosen to use it.

Nevertheless, there is one major problem with this implementation. It is only possible to superimpose on files, instead of sets of functions that are created based on annotations, naming conventions, directories and files. Solving this problem required a way to specify a super imposition unit. This is explained in the next subsection.

7.1.4 Superimposition unit

In the implementation described in the last subsection, it is only possible to superimpose on files. However, we identified that it should be possible to create sets of functions with files, annotations, naming conventions, and directories.

Creating a superimposition unit from a set of functions requires a manner to select functions, and add them to a concern. For realizing this we used a XML-file that

contains modules, with the properties: name, and a selector query, which results in a set of functions. This query can use all the prolog functions that can be used in the selector query of a composition filters concern.

In Listing 7.1 an example of such a XML-file is seen, where a superimposition unit CalcFunctions is created. The query selects all functions that start with the naming convention *"CALC"*.

```
<modules>
    <module name="CALC"
        query="Func|isFunction(Func),isFunctionWithName(Func, Name),
        matchPattern(Name,'CALC.*')"
/>
</modules>
```

Listing 7.1: `CConcern.xml with description of concern:`ClientFunctions

By defining these superimposition units, they can be used in the composition filters concerns, just as a file. We can select them with the selector element based on the functions they contain and/or its name, and use them as an external. With this extension to Compose⋆ C, it becomes possible to select sets of functions based on naming conventions, directories, and annotations. How to select the module with the selector is presented in Listing 7.2. This is the same as we proposed in Section 5.2.

```
    selectors
      moduleCALC = { C | isModuleWithName(C, 'CALC') };
```

Listing 7.2: `ModuleSelection` Selector with module selection

7.2 Collecting Program-Information

The static structure of the program is stored in the repository, which is a component of the language independent part of Compose⋆. The repository is a Very Abstract Syntax Tree(VAST), implemented as an object-oriented Database. In the repository not only program-information is stored, but also the information from the composition filters concerns. Storing the information from the composition filters concerns is done by the language independent part of Compose⋆, but retrieving the static structure information of the program is not.

In Compose⋆.NET type and method signatures are extracted from the input files in order to collect the program information. However, WeaveC contains a C parser that generates an Abstract Syntax Tree(AST) from the program. This AST is used for the weaving process that is done in WeaveC by adapting the AST and reconstructing a C-file from it. It offers all the static structure information needed for selecting join points. Because WeaveC and Compose⋆ are build in Java, the AST that is constructed from the source code, can be used to fill the repository.

The AST is constructed using ANother Tool for Language Recognition(ANTLR). ANTLR is a tool that can create recognizers, compilers, and translators from grammatical descriptions that may contain Java, C#, C++, or Python actions [47]. An ANTLR-program contains three elements:

- Lexer that contains the tokens of the language;
- Parser that contains the EBNF of the language to construct a tree;
- One or more treewalkers, which evaluate the tree.

To extract information from the program, the treewalker from WeaveC is adapted. In the treewalker actions are added, which add elements from the language-model to the repository, when they are visited in the tree.

Because annotations are not supported by C, they had to be added. This required adjustments to the parser and the treewalker. Annotations are first class entities, which can have arguments [45]. This is very similar to the structure element of C, by using typedef we can declare an annotation type. In this way we can control if the arguments of the used annotation are similar to the declared annotation type, which can reduce errors. In order to avoid problems with the current C syntax, an annotation starts with a '$'token, and is between '(' and ')'. An example of the possible programming elements where an annotation can be added is seen in Listing 7.3. There were only a few problems with the implementation of annotations:

File annotations
: In the listing one oddity is seen, $(file("annotated_file")). File is here a specific annotation type that is required because a file itself is not in the grammar of C. The program element to which an annotation is coupled is identified based on its position in the tree, but this is impossible for a file;

Composestar core
: Annotations in the language independent part of Compose⋆ only support one argument, which has to be of type String. Therefore, this is also only possible in Compose⋆ C.

```
1  $(file("annotated_file"))
2
3  typedef struct annotation_structure{
4      char* x;
5  } annotation;
6
7  $(annotation("annotated_global"))int man=3;
8
9  $(annotation("annotated_function")) void clientReceiveData
10             ($(annotation("annotated_parameter")) char* data)
11 {
12     printf("Client receiveData: %s \n", data);
13 }
```

Listing 7.3: `Possible` annotations

7.3 WeaveC

WeaveC is the aspect-oriented weaver that we selected for integration with Compose⋆ C, Chapter 6. WeaveC uses XML-files to specify the aspects. However, this is an unnecessary overhead, because WeaveC and Compose⋆ are both build in Java. WeaveC has a clear Java interface to specify the aspects in the program, which is used.

In Figure 7.3 the interface of an aspect is seen. This interface contains the following elements:

Aspect
: An aspect has an id and a series of pointcuts and advices.

Pointcut
: Consists of a name, a file, data, which makes it possible to select a specific element based on its name, a type, which is a function or struct, and an adviceapplication. With this information the join points can be selected.

AdviceApplication
: Is an element of pointcut, that is used to bind the advices and pointcuts, by using the same id's. It is also possible to declare the type of advice here, which can be before or after the pointcut.

Advice
: With its id it is possible to retrieve the adviceapplication, and therefore the pointcut, it also contains a type, a place where it should be woven, and a priority that indicates the order of the weaving of the advices. Advice also contains an element: code, which contains the actual behavior that needs to woven at the specific join point.

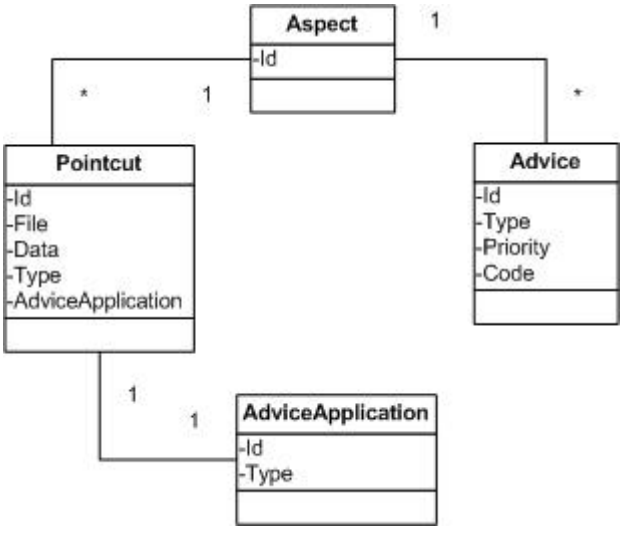


Figure 7.3: WeaveC aspect interface

This interface can be used to weave code in the base code. For this we need a mapping from the composition filters concern to this interface.

In Listing 7.4 a filter module is shown. On basis of this example we try to explain the mapping we implemented:

Aspect
A filter module self can be seen as an aspect, because it may contain a set of pointcuts where advice has to be added.

Pointcut
A pointcut is described with composition filters, by the matching part of the filter in combination with the selector of the superimposition. In this example the selector selects concern *ST*, the matching part of the filters selects [*.STpop], so function *pop* from concern ST is selected, [ST.*], all functions in file *Stack* are selected, and [*.*], all functions of concern *ST* are selected. Because WeaveC only supports weaving on the set of functions defined by a file, we need to retrieve the functions belonging to a concern and create a pointcut for every function separately.

AdviceApplication
The advice application needs to have an unique id for matching pointcut and advice, the id that is used is a combination of the filtername, matching function and advice application type. The advice application type depends on the filtertype used, for instance a prepend filter adds advice before a pointcut and an append filter after.

Advice
With its id it is possible to retrieve the adviceapplication and therefore this is the same combination of elements. The type of the advice depends if the filter is an input- or output filter. With an input filter the type is execution, while the type is call with an output filter. The priority is calculated by using the filtermoduleorder number. The code is dependent of the type of the filter in combination with a possible condition and substitution function.

```
1  concern ErrorConcern{
2    filtermodule error {
3      externals
4          st : Stack;
5      inputfilters
6        error_filter : Dispatch = {[*.STpop] *.ERprintErrorMessage,
7        [ST.*] *.ERprintErrorMessage,
8        [*.*] *.ERprintErrorMessage}
9      }
10   superimposition {
11     selectors
12       stackFunctions = { C | isModuleWithName(C, 'ST') };
13     filtermodules
14       stackFunctions <- error;
15   }
```

```
16 }
```

Listing 7.4: `Error filter module` for C

For matching part *.STpop, this example filter can be mapped to the aspect interface of WeaveC, this is shown in figure Figure 7.4.

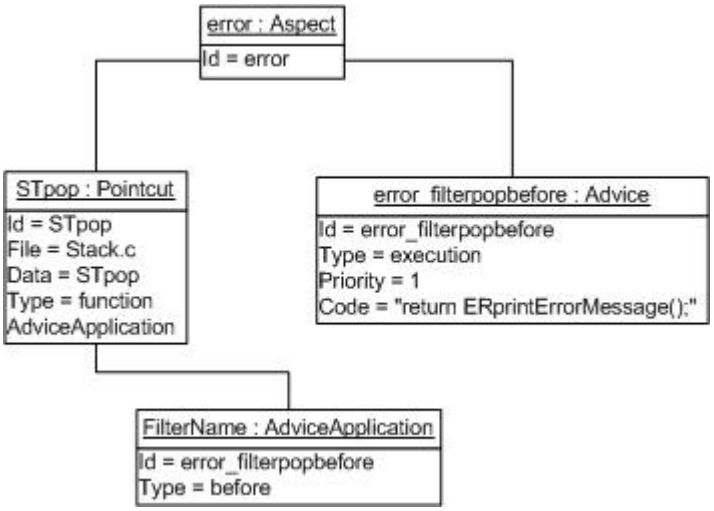


Figure 7.4: Filter module Error as a WeaveC aspect

In composition filters it is possible to redirect messages to methods in other files, or use conditions from files other than the matching function. In C this is only possible if there is a function declaration available in the file. Therefore, we implemented Compose⋆ C in such a way that when this is the case, an include statement is woven with the header file in which the function is declared. This means that it is required to create such a header. Preventing the include cycle problem can be done by using an *#ifndef*.

7.4 Filters

A filter is the element that is used to manipulate the behavior of a program. In Section 2.2 filters are explained. Here is seen that there are four built-in filter types, and that there is an option to create custom filters. In this section we explain the implementation of the four built-in filters: dispatch, send, error, and meta, and the custom filter.

7.4.1 Built-in filters

The implementation of the filters is the advice that has to be woven. In composition filters we can make a distinction between the built-in filters: dispatch, send, and error,

which are well defined and therefore possible to reason about. This is not possible for Meta filters, because they are not well defined. However, meta filters offer a lot of functionality. Therefore Compose⋆ C offers support for both.

7.4.1.1 Dispatch filter

The dispatch filter redirects an accepted message to a specified target. Otherwise the message continues to the subsequent filter.

Because, we proposed that a message is a function call, it is possible to redirect messages by calling the substitution function from inside the matching function. And returning the value of the substitution function, when control is given back to the matching function. In 7.5 an example of a dispatch filter, Listing 7.5, and the generated code, Listing 7.6, is presented.

```
1 inputfilter
2     dispatch_filter : dispatch = { condition => [*.foo] *.bar }
```

Listing 7.5: `Dispatch` input filter

```
1 int foo(){
2         if(condition()==1){ return bar();}
3         ..............
4 }
```

Listing 7.6: `Dispatch input filter advice`

Dispatch filters do not have an output filter implementation.

7.4.1.2 Error filter

The error filter raises an exception if the filter rejects the message, otherwise the message continues to the next filter in the set.

In C there is not a standard way for handling exceptions as for instance in Java. However, the standard C library offers a view approaches that manage exceptions. These approaches all handle a set of stages in the exception's lifetime. These exception stages are:

1. A software error occurs;
2. Capturing an error's cause and nature in an exception object;
3. Detecting the exception object;
4. Handling the exception;
5. After handling the exception, the program recovers and continues. execution.[53]

The approaches are [53]:

stdlib.h
: Provides the functions that halt the program: *abort*, abnormal program destruction, and *exit*, civilized program destruction by calling registered handlers. Abort and exit cause an absolute termination of the program. These functions implement exception lifetime Stages 4 and 5.

assert.h
: The *assertion* macro is an abort function with a preceding condition check and generated feedback that shows the source file and line number. It implements exception Stages 3 through 5. Assertions are mainly used for debugging conditions that can never arise in a program.

setjmp.h
: *Longjmp* in combination with *setjmp* implement exception lifetime Stages 2 and 3. Setjmp(j) sets a jump-point by filling j with the program context. When setjmp is called it returns 0, therefore it can be used in a condition-expression. By calling longjmp(j, r) a non-local goto to jump-point j is invoked. The setjmp function now returns value of r, or when not set 1. Now the condition-expression can invoke an exception.

signal.h
: Contains a *signal* and *raise* function. With these functions it is possible to intercept signals and pass them on to a manually defined handler, which for instance can implement an abort *function* or *longjmp* for handling the exception. However, there are only a few conditions in a C program that may rise a signal, for instance division by zero, and memory access violations. Handling exceptions with signal and raise implements all stages of an exception's lifetime, nevertheless it only intercepts a few exceptions.

errno.h
: This header files declares a *errno* external and several values, that identify the type of error, it may take. The value of errno has to be initialized to 0 at the beginning of a program. Errno is set by some standard libraries when an error occurred. It is also possible to change the errno value in your own functions. If after such a function call the value of errno has one of the defined values a specific exception has to be raised. Errno.h implements exception stages 1 to 3.

In composition filters an error has to be raised when a condition in combination with the condition operator is false. This means that it does not have to intercept software errors, and to detect its cause and nature, stages 1 and 2. It only has to handle the exception and recover the program, and possibly notify where the error is raised. Therefore the best approach for composition filters is, to use assertions, because it evaluates a condition, if false it generates information about the file and line number, and terminates the program.

The advice that has to be generated for an input and output error filter is simple, Listing 7.8. An assertion call requires a condition as argument and it automatically

returns from the program when the condition is false. Therefore an assert call for an input error filter can be woven at the execution of the function, line 4, and for the output error filter for all calls from this function, line 6. Because the assert function is defined in a header, the header should be included in the file the advice is added, line 1.

```
1 inputfilter
2     error_execution_filter : error = { conditionInput => [*.foo] }
3 outputfilter
4     error_calling_filter : error = { conditionOuput => [*.bar] }
```

Listing 7.7: `Error` filter

```
1 #include <assert.h>
2
3 int foo(){
4         assert(conditionInput());
5         ..............
6         assert(conditionOutput());
7         bar();
8         ..............
9 }
```

Listing 7.8: `Error filter advice`

7.4.1.3 Send filter

If the message is accepted, it is send to the specified target of the message, otherwise the message continues to the subsequent filter.

The send filter can only be used as an output filter. Output filters intercept messages at the call join point of a function. In the example presented in Listing 7.9 the behavior of the send filter is shown. The woven code at lines 3 and 4, shows that after the if-statement the *foo()* call has to be caught within an else-statement.

```
1 outputfilter
2     send_filter : send = { condition => [*.foo] *.bar }
```

Listing 7.9: `Send` filter

```
1 int afunction{
2     ....................
3     if(condition()==1){bar();}
4     else{
5         foo();
6     }
7     ....................
8 }
```

Listing 7.10: `Send filter advice`

Nevertheless, generating this code offers some problems:

Does not pass the treeparser

All the code that has to be woven is checked if it is perform the C syntax. This is done by generating an AST of the advice, adding it to the tree of the file and parsing the new subtree. Weaving a not closed else-statement is not allowed by the parser, lines 3, and 4, of Listing 7.10.

Return value

If the return value of the matching function is used, for instance to update a variable, this also shall not pass the parser. This is seen in Listing 7.11, line 3, where the behavior is added between *store=* and the *foo()* call.

```
1         int afunction{
2                 int store;
3                 store=if(condition()==1) bar(); else foo();
4                 ......................
5         }
```

Listing 7.11: `Return value` problem

Because of these reasons, generating this code for a send filter is not optionally. Generating the same advice as the input dispatch filter, solves these problems. However, every time the function is called the advice is executed, instead of only the times when the function is called from the selected concern. This problem can be solved by setting a boolean value before the actual call from the right functions, as in Listing 7.12.

```
1  int fooCalled=0;
2
3  int afunction(){
4      ......................
5      fooCalled=1;
6      foo();
7      ......................
8  }
9
10 int foo(){
11     if(fooCalled==1){
12         if(condition()==1){ fooCalled=0; return bar();}
13     }
14     ..............
15 }
```

Listing 7.12: `Send` filter with control flow information

This implementation solves the problems. However generating this code, requires three weaving operations instead of one, lines 1, 5, 11, and an extra delay of the program every time the function is called, line 11. In the current version of Compose⋆ C send filters are not implemented in this way. Therefore send filters do not work in combination with a condition, and also not if the calling function is used inside an expression.

7.4.1.4 Meta filter

If the message is accepted, the reified message is sent as a parameter of another -meta message- to a named object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message, then re-activate its execution.

A reified message may contain diverse information about the message, for instance arguments, target, selector, and sender. For Compose⋆ C we have chosen for a minimal reified message that passes argument information, the type as well as its value, and the matching function's name. Grouping arguments in a reified message makes it possible to redirect messages from functions with different arguments toward the same function. This can be convenient with *Logging* or *Tracing* concerns.

In Compose⋆.NET also sender information is stored in the reified message. Sender information offers runtime information, which is seen in other aspect-oriented languages in the form of join points, such as ControlFlow and Within. However, retrieving the sender at compile time is only possible by using a global variable that saves the information before every function call. This is similar to the principle from example Listing 7.12, only we have to do this for every function call in the program and save sender information instead of setting a boolean.

The reified message should be passed as an argument to the function where the message is redirected to. We have chosen to implement the message as a struct, with a character pointer type that uses the same characters as printf to define the type, an int with the number of arguments, and a void pointer to an array, which saves the argument values. Type and arguments are pointers, because the struct always has to have the same size. We also used a fixed size array, because in the older GCC version this was mandatory. The message struct is declared in the file message.h that comes with the Eclipse-plugin and is automatically added to the file, in which the matching function of the filter is declared. Message.h is presented in Listing 7.13, the combination of type and argument is clarified in Figure 7.5.

language

```
1  #ifndef COMPOSESTARC_MESSAGE_H
2  #define COMPOSESTARC_MESSAGE_H
3
4  struct message{
5      char* type;
6      int argumentNumber;
7      void* argument[100];
8  };
9
10 #endif
```

Listing 7.13: `Message.h`

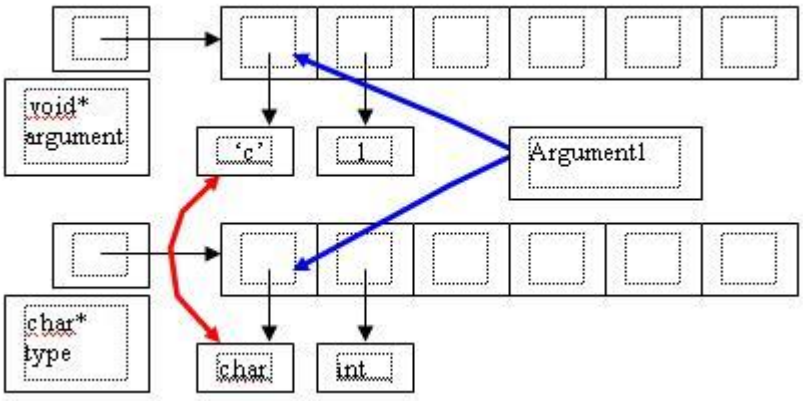


Figure 7.5: Storing argument information

A characteristic of the meta filter is that it supports different reactivation actions. These actions determine the manner of returning to the matching function. The possible actions are proceed, resume, and return. An example of the proceed action call is seen in Listing 2.3. The implementation of these actions is based on multiple threads. There are different threads for target, filterset, sender, ACT, where the return stops the filterset, and returns to the target, the proceed goes to the target function and later resumes the ACT, and the resume stops the ACT and goes to the next filter [57]. We do not prefer the multiple thread implementation for Compose⋆ C, because it reduces the efficiency of the program.

With a little difference in behavior, it is possible to implement the reactivation actions as in Listing 7.14, Listing 7.15, and Listing 7.16. Nevertheless, it is impossible to select one of these different behaviors at compile-time. Furthermore, the proceed meta filter implementation is not without errors. We have chosen to implement the resume meta filter behavior in Compose⋆ C, because this behavior is needed for Tracing and Logging concerns. It is possible to create the same behavior as the proceed and return meta filter by using custom filters.

```
1   int firstTime=1;
2
3       int bar(message msg){
4           .......
5           firstTime=0;
6           int returnvalue = foo();
7           .......
8           return returnvalue
9       }
10
11     int foo(){
12              if(firstTime==1){
13                  struct message msg;
14                  msg . argumentNumber = 0;
```

```
15                      msg . type = "e" ;
16                      if(condition()==1){bar(&msg);}
17                  }
18                  ..............
19          }
```

Listing 7.14: `Proceed action of a meta` filter

```
1     int foo(){
2           struct message msg;
3               msg . argumentNumber = 0;
4               msg . type = "e" ;
5               if(condition()==1){ bar(&msg);}
6               ..............
7           }
```

Listing 7.15: `Resume action of a meta` filter

```
1       int foo(){
2           struct message msg;
3               msg . argumentNumber = 0;
4               msg . type = "e" ;
5               if(condition()==1){return bar(&msg);}
6               ..............
7           }
```

Listing 7.16: `Return action of a meta` filter

7.4.2 Custom filter

With custom filters it is possible to create specific behavior that should be added to the base program. In order to use custom filters, there should be a manner to flexibly add them to Compose⋆, and a simple way to create a custom filter; for defining it, as well as making use of base program information.

A way to flexibly load classes in Java, is by using Java's class loader. The class loader makes it possible to look for compiled Java classes with a specific name, which are not in the project. In this way we are able to define the custom filters in Java and to use information from the repository. By letting this custom filter class extend an abstract class, Semantic, it is possible to create a specific interface for the custom filter classes and create some syntactic sugar for easily retrieving program information. There is one drawback to this implementation, which is that the C-programmer needs to have some knowledge of Java in order to create custom filters.

As we have seen with the built-in filters, the filters all have their own behavior and specific characteristics. These characteristics are:

Type
: The name of the filter as it is used in the composition filters concern. This name should be the same as the name of the class, because it is used by the class loader;

Before advice
The advice that has to be woven before the join point;
After advice
The advice that has to be woven after a join point;
Redirect messages
Some filters make use of a substitution target, while others do not;
Header file
By specifying a header file name here, this header will be included in the file of the matching function. This is sometimes needed, for instance when a message should be redirected.

More information about the custom filter is described in Appendix C, where is shown how to make one, and as an example a decrypt custom filter is presented.

7.5 Integrated Development Environment

An Integrated Development Environment(IDE) is a combination of several tools that are used in a software development process. Early IDE's connected an editor, compiler and debugger to expedite program development. Nowadays, the supported features of an IDE are very widespread [13]. IDE's improve overall ease-of-use and performance of developing new programs.

In Section 3.4 we stated that the usability of the Compose⋆ C is important, because we want it to be applicable. Integrating it with an IDE will improve Compose⋆ C's usability. Extensive IDE's offer for instance a user-interface, filebrowser, text-editor, compilers, debugger and are expendable. When such IDE is extended with Compose⋆ C, Compose⋆ C can use these features.

Compose⋆.NET is an extension of Microsoft Visual Studio .NET which is probably the most popular IDE, because it is used for developing applications for Microsoft Windows. In Figure 2.3 is seen that the IDE creates a build configuration that is used by the compile-time of Compose⋆. The build configuration is generated by Visual Studio that automatically derives most of the settings from the project. For settings which cannot be generated an interface is supported. It is also possible to run a Compose⋆ application from inside Visual Studio.

Unfortunately, Visual Studio only runs on Windows, while C is very popular among other platform users, such as Linux. Therefore a plug-in for another IDE is developed, being Eclipse. Eclipse is a free IDE, built in Java and therefore platform independent, and it is easy to extend. There are many plug-ins developed for Eclipse, such as support for C, which is not hard-coded in Eclipse. Eclipse is probably the second most popular IDE.

The Eclipse-plugin solves two issues that make Compose⋆ C unpractical for use, being:

- Complex build configuration file;
- Intermediate steps in the building process of the program.

These issues are explained more extend in the next paragraphs, just as the implementation of the plug-in. The plug-in is also designed to work with Compose★ Java, therefore it includes some useless functions for Compose★ C. The plug-in is presented in Appendix F.

7.5.1 Build configuration

Compose★ requires a build-configuration file in order to compile a project. This build-configuration is an XML-file that defines the sources, concerns, paths, settings and compiler information. A build-configuration file is shown in Appendix A.

Every project we want to create with Compose★ C needs a build-configuration. Nevertheless, the build-configuration file is not:

Reusable
: It is project-dependent, except for the platform part;

Adaptable
: It is easy to make mistakes when this file is manually changed.

However, reusing the file and trying to adapt it is probably easier then rewriting it in total. Also the build-configuration needs to be changed sometimes, for instance when the debuglevel has to be changed or a new file is added to the project.

Creating or adapting a build-configuration file manually is error-sensitive and time-consuming. This negatively affects the usability of Compose★ C, therefore automatic generation of such a file is implemented.

The sources, concernsources, composestar-path and classpath, and the language found in the buildconfiguration file can be derived from a project as it is created in Eclipse. The debuglevels, applicationStart, basepath, buildpath, outputpath, and Compose★ module properties cannot be retrieved, therefore a possibility to set them from inside Eclipse is created.

7.5.2 Building process

For building a C-program a series of actions need to be executed. First the source-files are preprocessed, then compiled, assembled and finally linked with the other source files in the program. Normally, a C-compiler automatically executes these steps sequentially. However, creating a program with Compose★ C requires changes to this building process.

We selected WeaveC for realizing the weaving. WeaveC requires preprocessed source-files and aspects in order to apply the weaving. Redirecting messages with a composition

filters concern requires weaving of an #include statement. This is a preprocessing directive, therefore after weaving the preprocessor has to be called again. The included files should still be available to the preprocessor, so when the Compose⋆ C output is placed in another folder, these files should be copied to the folder as well. By using WeaveC intermediate steps are needed in the building process of the program.

A program build with Compose⋆ C requires a library where the reified message is defined and some functions that use this message. This library has to be added to the project when a meta filter, or custom filter is used.

For building a C-program the program *make* is often used. Make interprets various rules defined in a makefile that for instance specify options for the compiler, files that need to be compiled, dependencies between files, or other programs that need to be called. Such a makefile can also be created to carry out the building process of Compose⋆ C as seen in Listing 7.17. Calling the target *build* from the makefile, 48, will successively perform the actions: *copyMessageFromPlugin*, *preprocess*, *weave*, *copymessage*, *copyhfiles*, *changeextensions*, *compile*, *clean*, *run*.

The output of the preprocessor and Compose⋆ C changes the extensions of the files, otherwise it is possible that the original files are overwritten. However GCC does not accept files with an extension: *.ccc.out*. Therefore target *changeextensions*, 30, is added.

Just as the build-configuration file, creating a makefile manually is error-sensitive and time-consuming, which reduces the usability of Compose⋆ C. Therefore, automatic generation of the makefile is supported.

This is possible, because all information is retrievable. The paths and concerns are available in the build-configuration file and the execution sequence is always the same.

```
1  copyMessageFromPlugin:
2      $(fileCopy)
3          c:\Eclipse\plugins\CstarEP\message.*
4          H:\EncryptionExample\
5
6  preprocess:
7      gcc -dD -E Server.c -o Server.ccc
8      gcc -dD -E Encryptor.c -o Encryptor.ccc
9      gcc -dD -E message.c -o message.ccc
10     gcc -dD -E Logger.c -o Logger.ccc
11     gcc -dD -E Protocol.c -o Protocol.ccc
12     gcc -dD -E ExampleRunner.c -o ExampleRunner.ccc
13     gcc -dD -E Client.c -o Client.ccc
14     del message.ccc
15 weave:
16     java.exe  -cp
17         "c:\Eclipse\plugins\CstarEP\binaries\antlr.jar$(SEP)
18         c:\Eclipse\plugins\CstarEP\binaries\prolog.jar$(SEP)
19         c:\Eclipse\plugins\CstarEP\binaries\ComposestarC.jar$(SEP)
20         c:\Eclipse\plugins\CstarEP\binaries\ComposestarCORE.jar$(SEP)"
21         Composestar.C.MASTER.CMaster
22         "H:\EncryptionExample\BuildConfiguration.xml"
```

```
23
24 copymessage:
25     $(fileCopy) message.c H:\EncryptionExample\bin\
26
27 copyhfiles:
28     $(fileCopy) *.h H:\EncryptionExample\bin\
29
30 changeextensions:
31     $(fileCopy)
32         H:\EncryptionExample\bin\*.cccout H:\EncryptionExample\bin\*.c
33
34 compile:
35      gcc -oH:\EncryptionExample\bin\ExampleRunner.exe
36        H:\EncryptionExample\bin\Server.c
37        H:\EncryptionExample\bin\Encryptor.c
38        H:\EncryptionExample\bin\Logger.c
39        H:\EncryptionExample\bin\Protocol.c
40        H:\EncryptionExample\bin\ExampleRunner.c
41        H:\EncryptionExample\bin\Client.c
42        H:\EncryptionExample\bin\message.c
43
44 run:
45     H:\EncryptionExample\bin\ExampleRunner
46
47 build: copyMessageFromPlugin preprocess weave
48         copymessage copyhfiles changeextensions compile clean run
```

Listing 7.17: `makefile` for Compose⋆ C

7.6 Summary

During the design phase of Compose⋆, the possibility that Compose⋆ would be ported to other platforms was already taken into account, which resulted in a clear separation of the platform independent and dependent part of Compose⋆. The independent part is usable for all the platforms, while the platform dependent part should be replaced by a specific implementation for the language to which Compose⋆ is ported.

In this chapter we discussed the Compose⋆ implementation for C, or more specifically the implementation of its platform dependent part. The platform dependent part consists of five important elements, being:

Language Model
: We used the abstract language model of the language independent part of Compose*, where we implemented a file as a class, a function as a method, and a directory as a namespace. We furthermore created a possibility for defining concerns in a specific file, as a set of functions, specified by a selector query;

Collecting program-information
: WeaveC creates an AST from the source code. Because the repository is a VAST,

which resembles an AST, we used a treewalker to extract the information from the AST and store it in the repository. The original grammar supported by WeaveC did not include annotations, so we extended the grammar with them;

WeaveC
: For defining aspects WeaveC uses XML. However, WeaveC and Compose⋆ are both developed in Java, therefore it is possible to directly instantiate aspect objects and skip creating intermediate-files. We defined a mapping from the filters to the interface of the aspects. The aspects are automatically created by Compose⋆;

Filters
: This is the actual advice that is added to the base program. We described the behavior of the built-in filters of Compose⋆ in C, and created a possibility for implementing custom filters. We changed the behavior of meta filters slightly, because we did not like the original implementation that used multiple threads;

Integrated Development Environment
: To enhance the usability of Compose⋆ C an Eclipse plug-in is created. This plug-in generates a build-configuration file, and simplifies the building process of the application under construction.

The architecture of Compose⋆ C is seen in Figure 7.6.

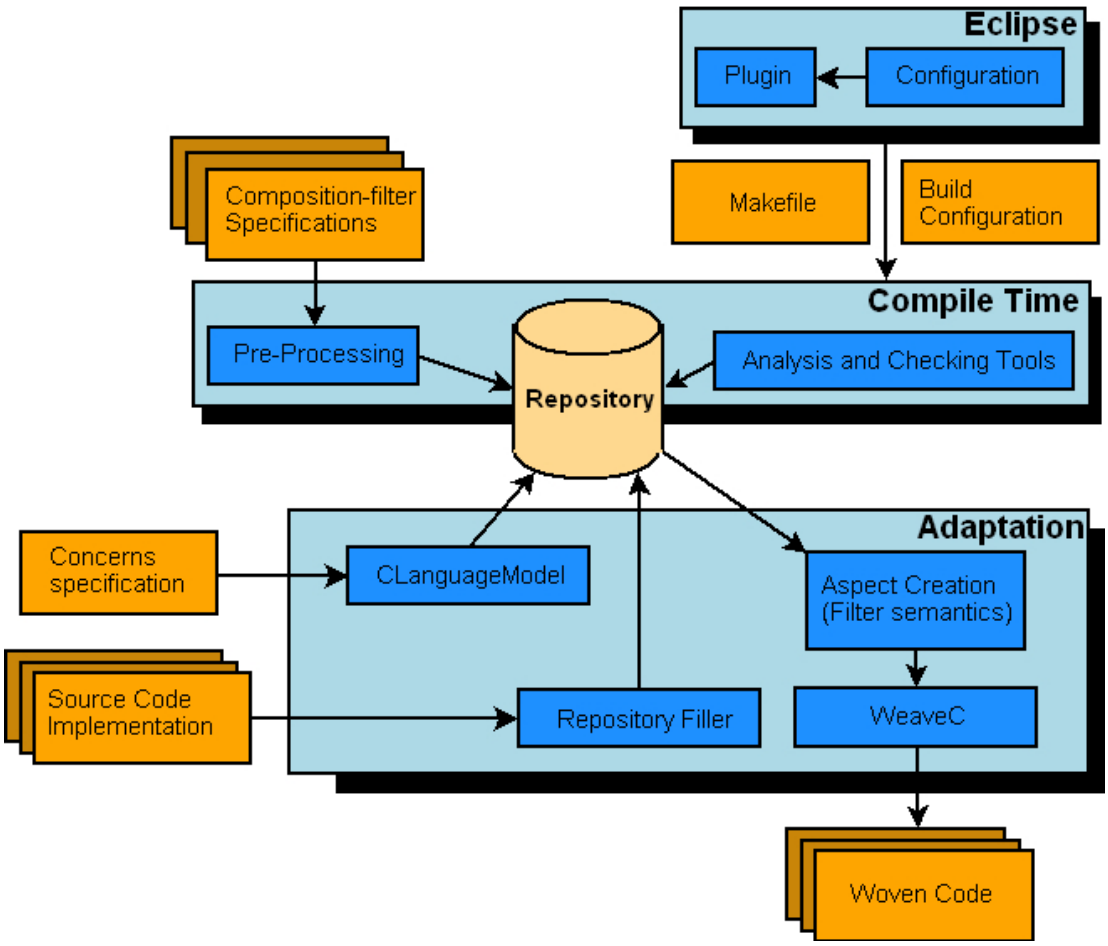


Figure 7.6: Compose⋆ C Architecture

Chapter 8

Conclusion

In this thesis we have shown that composition filters can be applied to C. We proved this by implementing our idea, how composition filters should be applied to C, with as a result Compose⋆ C. In this final chapter we summarize our findings, evaluate if the design fulfills the requirements we formulated, show some points of improvement, and briefly compare the design with existing aspect-oriented weavers for C.

8.1 Summary

Compose⋆ C is an aspect-oriented weaver for C based on the composition filters approach. It is developed as a solution to crosscutting problems in C applications. Central to this research were the problems found in the system of ASML. The code of this system is difficult to change, harder to reuse, and harder to understand, because of four major crosscutting concerns: reflection, dynamic execution tracing, parameter checking, and timing. Making it possible to weave these crosscutting concerns back into the base code can be seen as the functional requirement of Compose⋆ C. We also formulated some quality requirements, which we deduced from the characteristics of C and the intention of composition filters. We would like to see that Compose⋆ C becomes generally applicable, efficient, and compatible with other Compose⋆ concerns.

Because composition filters is an extension to the object-based model, and C is a procedural language, which does not support all the object-based program elements, a mapping of composition filters in C has been made. This mapping describes that a message can be seen as a function call, or an update of an external variable. Updating an external variable is nevertheless not implemented. Messages are intercepted by input- and output-filters, where input filters intercept messages at the execution join point, and output filters intercept them at the call join point. The superimposition unit of composition filters, is a class in the object-based languages, is in C a function, or set of functions that can be defined by a file, directory, naming convention, and/or annotation.

Compose* is an aspect-oriented weaver based on the composition filters approach for Java and .NET. The design of Compose* is partly language dependent, and partly language independent. Porting Compose* to a new programming language requires a new implementation of the platform dependent part. For Compose* C we needed to implement a language-weaver, the C language model, a component that collects program information, filters, and IDE support. However, implementing a weaver is a time consuming job, so we selected an existing weaver, WeaveC, and integrated it in Compose* C.

8.2 Evaluation

In this research the crosscutting concerns of the ASML system were central. Separating these concerns is therefore considered as the functional requirement for Compose* C. Furthermore, we proposed a few quality requirements. In this section we shall evaluate if these requirements are met by Compose* C.

8.2.1 ASML concerns

The crosscutting concerns in ASML's system are reflection, dynamic execution tracing, parameter checking, and timing. Listing 8.1 presents the example without the crosscutting concerns. The example with the crosscutting code is shown in Listing 3.1. In this function annotations are used for specifying parameters. In Listing 8.2 the example function is shown again, after the concerns are woven with Compose* C.

In the *calcDivide* function of Listing 8.2 all crosscutting concerns are present. For testing purposes the actual advice is slightly adjusted, but the join points at which the advice is added are the same. The dynamic execution tracing advice is shown in lines 28 and 55. The parameter checking code is found at line 46. And the timing advice is shown in lines 24 and 59. The advice for the reflection concern, the *function_name* variable, is now directly used by other concerns, such as in line 20 for the timing concern.

In the original code the concerns also used some global variables which are available after preprocessing. The timing concern variables are declared in the include file *"Timing.h"* line 11.

From this example we can see that Compose* C is able to separate the concerns from the base code. So we may conclude that Compose* C fulfills the functional requirements and addresses the problems caused by crosscutting concerns. Because, Compose* C enhances the reusability of the original functions, since they do not contain crosscutting code anymore. It improves the changeability of the code, because the concerns are not scattered or tangled anymore. And it also improves the understandability of the system, which becomes clear when Listing 8.1 and Listing 8.2 are compared. Furthermore, it seems that the size of the program can be reduced, because there are only three small

concerns needed to weave the crosscutting code back into the base-code. Nevertheless, we did not have the complete source and therefore we did not test the actual code reduction.

```
#include "Calc.h"

struct annotatie{
    char* x;
};

typedef struct annotatie in;
typedef struct annotatie out;
typedef struct annotatie inout;

int calcDivide($(inout(""))double* answer,
        $(in(""))double op1, $(in(""))double op2)
{
    int result=OK;

    if(op2 != 0)
        *answer = op1 / op2;
    else
        result =ERROR;

    return result;
}
```

Listing 8.1: `Original code` calcDivide

```
struct annotatie {
    char * x ;
} ;

typedef struct annotatie in ;
typedef struct annotatie out ;
typedef struct annotatie inout ;

#include "Error.h"
#include "message.h"
#include "Timing.h"
#include <assert.h>

int calcDivide ( double * answer , double op1 , double op2 )
{
    int result = 2 ;
#line 16 "Calc.c"

    {
        time_t time1 ; char * function_name = "calcDivide" ;
        ( void ) time ( & time1 ) ;
        printf ( "\n Timing Concern:%s \n" , function_name ) ;
        TIMING_in ( function_name , time1 ) ;
    }
    {
```

```
26          printf ( "\n Tracing Concern:calcDivide \n" ) ;
27          TRACE_in ( "calcDivide" ) ;
28      }
29      {
30          printf ( "ParameterChecking Concern:calcDivide " ) ;
31          if ( answer == 0 )
32          {
33              printf ( "InOut parameter in function =
34                  NULL function: calcDivide parameterName= answer \n");
35          }
36          if ( op1 == 0 )
37          {
38              printf ( "Input parameter in function = NULL function:
39                  calcDivide parameterName= op1 \n" ) ;
40          }
41          if ( op2 == 0 )
42          {
43              printf ( "Input parameter in function = NULL function:
44                  calcDivide parameterName= op2 \n" ) ;
45          }
46      }
47      if ( op2 != 0 )
48      * answer = op1 / op2 ;
49      else
50      result = 3 ;
51  #line 21 "Calc.c"
52
53      {
54          printf ( "\n Tracing Concern:calcDivide \n" ) ;
55          TRACE_out ( "calcDivide" ) ;
56          {
57              time_t time2 ; ( void ) time ( & time2 ) ;
58              printf ( "\n Timing Concern:calcDivide \n" ) ;
59              TIMING_out ( "calcDivide" , time2 ) ;
60              return result ;
61          }
62      }
63  }
```

Listing 8.2: Woven code calcDivide

8.2.2 Quality requirements

In Section 3.4 we stated three requirements that we want to see fulfilled by Compose* C. In this section we evaluate if this is the case.

General applicable

Compose⋆ C can be used to add concerns to existing C applications. However, we used the C compiler of WeaveC, which works on preprocessed source code. Preprocessed code is dependent of the C compiler used, for instance newer C compilers generate more compiler optimization commands. Because other compilers do not know these commands, the preprocessed code cannot be parsed with every C compiler. This is also the case for Compose⋆ C that works with the compiler GCC-2.95. In principle, this is not a big problem, because most C programs can also be compiled with an older C version. However, there are some options in the newer GCC versions that are not supported by GCC-2.95, for instance variable sized arrays. If such options are used, using Compose⋆ C requires some adjustments to the application.

Another part of this requirement was that we did not want to make Compose⋆ C dependent from naming conventions. With Compose⋆ C there are several ways to define a concern. It can be a single function, but it can also be based on naming conventions, annotations, files, and directories.

Efficient

The woven advice of Compose⋆ C is normal C code, so the woven program can still be compiled with a normal C compiler. We have seen in Listing 8.2 that the original code can be reconstructed, and even the C compiler optimization is still possible. Because of this the application preserves its efficiency. Nevertheless, using Compose⋆ C increases the compilation time of an application. In Table 8.1 some statistics of Compose⋆ C are presented, which are acquired by testing the programs HelloWorld, a small program existing of two files and one filter, and EncryptionExample, a program consisting of six files and four filters. In the table a row *Weave* is added that shows the actual performance of Compose⋆ C, so without preprocessing and compiling with GCC, and the renaming and copying of files. The timing results of GCC are presented in Table 8.2. From the testing results we can conclude that the total time that Compose⋆ C requires to compile and run a program is approximately 4x slower than only using GCC.

	HelloWorld	EncryptionExample
Built + run	2,8	5,2
Weave	1,7	3,3
Run	0,1	0,1

Table 8.1: Timing results of Compose⋆ C

	HelloWorld	EncryptionExample
Built + run	0,7	1,3
Run	0,1	0,1

Table 8.2: Timing results of GCC

Compatibility of Compose⋆ concerns

Nevertheless, not all the elements of a composition filters concern are still usable, therefore not all concerns of Compose⋆.NET and Compose⋆ Java are applicable to Compose⋆ C. The internal element of the filter module part and the annotations element of the superimposition part are not implemented, because they do not really fit the concept of C. Maybe this requirement was not really realistic, because in the external and internal elements constructors of the objects are called. Calling a constructor does not exist in a procedural programming language. However, we tried to make the concerns of the different ports as compatible as possible, therefore the selector language of Compose⋆ C does support object-based elements as well as C elements, for instance files.

8.3 Future work

For this research, the implemented functionality of Compose⋆ C is sufficient. However, there are some possible extensions which could be useful. These extensions are:

Composition filters elements
: Composition filters is an extensive paradigm, because some features are not very useful for this research, we did not implement them. These features are: introduction of annotations, and constraints.

WeaveC
: At the moment only the needed features of WeaveC are used in the implementation of Compose⋆ C. However, WeaveC does support more functionality that can be useful, such as introducing local variables and elements in structures. This extension will make it possible to add a local variable before the function execution join point, and use it in the advice that is added before and after a function execution. At the moment variables that are declared in the advice, which is woven before a function execution, is added between brackets, and therefore in a separate scope.

Composition filters in C
: In our design of composition filters in C we identified that updating a global variable can be viewed as a message. Nevertheless, we did not implement the interception of these messages, so we cannot add filter modules to it. Furthermore, it is sometimes impossible to weave output filters at a specific join point, because the function call is for instance inside an expression. With some changes to the weaver of Compose⋆ C this problem can be solved.

When creating an application with Compose⋆ C, there are some features that should be accounted for. These features can possibly be improved:

Header files
Conditions or functions, to which we want to redirect messages, should be declared in a header file that has the same name of the file they are defined in. In this way, the header can automatically be included in the woven file.

Annotations
Using annotations in the base code means that a normal C compiler cannot parse it anymore, only Compose⋆ C can. You possibly can rename annotations by using the *-D$(..)=""* flag when calling GCC, but this option is not supported in the makefile, because we are not sure if there are any side effects.

8.4 Related work

In Chapter 6 we already evaluated the current C weavers based on the requirements needed for implementing composition filters. In that chapter we saw that there are only five weavers which actually work. Nevertheless these weavers all have their shortcomings, being:

AspectC(06)
is a very complete weaver with many pointcuts. Nevertheless, it does not have any support for introducing globals or including headers, ordering advice, and using annotations.

Arachne
does not fulfill the requirement of general applicability, because it only works on a computer with an IA32 processor and a Linux distribution that supports anonymous memory mapping;

AspectC++
does not fulfill the requirement of general applicability, because it only works on C++ instead of C;

Aspicere
its compilation process is very slow, it does not support an execution join point, and also has no IDE support, but it can be used on different C versions;

WeaveC
fulfills the requirements for the weaver, and therefore it is integrated in Compose⋆ C. Nevertheless, in WeaveC aspects are defined in XML, which is not user-friendly.

As we already saw in Chapter 6, only WeaveC fulfilled all the requirements stated in our research. Compose⋆ C improves the usability of WeaveC, but this also has a consequence for the compilation time. In our opinion the increased usability results in a bigger saving of time than the extra compilation time, which is needed for one extra preprocessor run and the evaluation of Compose⋆. Because of this we believe Compose⋆

C to be an improvement over WeaveC, and the other aspect-oriented weavers for C. Furthermore, the reasons for choosing the Composition Filters approach instead of the AspectJ approach were the high-level of semantics and the analyzable aspect description language, which enhances the usability of the program. These features still apply, which is another improvement over the current aspect-oriented weavers for C.

8.5 Concluding

With Compose⋆ C it is possible to reduce the problems that are caused by crosscutting concerns. Compose⋆ C fulfills all the requirements needed to weave the ASML crosscutting concerns back into the base code. Nevertheless, not all the quality requirements are fully met, and there still are some enhancements possible for Compose⋆ C. But we can conclude that Compose⋆ C already is a very usable tool and an improvement over the current aspect-oriented weavers for C.

Bibliography

[1] B. Adams. Aop on the c-side. *In Proceedings of the 2nd Linking Aspect Technology and Evolution Workshop (LATEr)*, 2006.

[2] B. Adams and T. Tourwe. Aspect orientation for c: Express yourself. In *AOSD '05: In 3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT)*, 2005.

[3] B. Adams, K. D. Schutter, and A. Zaidman. Applying aspects in a legacy environment, 2005.

[4] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. page 184, 1997. IBM Centre for Advanced Studies.

[5] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. page 4, 1998.

[6] Arachne, July 2006. URL http://www.emn.fr/x-info/arachne/.

[7] AspectC++, July 2006. URL http://www.aspectc.org.

[8] AspectC, November 2006. URL http://www.aspectc.net/.

[9] Aspicere, July 2006. URL http://users.ugent.be/~badams/aspicere/.

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

[11] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [32], pages 63–95. ISBN 0-321-21976-7.

[12] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems migration: A brief review of problems, solutions, and research issues. Technical report, Computer Science Department, Trinity College, Dublin, 1999. Technical Report TCD-CS-1999-38.

[13] C. Boekhoudt. The big bang theory of ides. *ACM QUEUE*, page 74, 2003.

[14] S. R. Boschman. Performing transformations on .NET intermediate language code. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[15] R. Bosman. Automated reasoning about Composition Filters. Master's thesis, University of Twente, The Netherlands, Nov. 2004.

[16] M. Brody and M. Stonebraker. Migrating legacy systems: Gateways, interfaces and the incremental approach. 1995.

[17] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. 2005.

[18] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. Linking analysis and transformation tools with source-based mappings. 2006.

[19] C programming language, februari 2006. URL http://en.wikipedia.org/wiki/C_programming_language. Wikipedia.

[20] C4, July 2006. URL http://c4.cs.princeton.edu/.

[21] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643609.

[22] S. Comella-Dorda, K. Wallnau, R. C. Seacord, and J. Robert. A survey of legacy system modernization approaches. Technical report, 2000. Technical Note CMU/SEI-2000-TN-003.

[23] O. Conradi. Fine-grained join point model in Compose*. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[24] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[25] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052901.

[26] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. Sdholt. An expressive aspect language for system applications with arachne. *Transactions on Aspect-Oriented Software Development*, 2006.

[27] P. Durr, L. Bergmans, G. Gulesir, and I. Nagy. Towards an expressive and scalable framework for expressing join point models. 2005.

[28] P. E. A. Dürr. Detecting semantic conflicts between aspects (in Compose*). Master's thesis, University of Twente, The Netherlands, Apr. 2004.

[29] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10):29–32, Oct. 2001.

[30] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052903.

[31] M. Engel and B. Freisleben. Using a lowlevel virtual machine to improve dynamic aspect support in operating system kernels. In *4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2005.

[32] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[33] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

[35] 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.

[36] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. ISBN 0471431044.

[37] W. Havinga. Designating join points in Compose* - a predicate-based superimposition language for Compose*. Master's thesis, University of Twente, The Netherlands, May 2005.

[38] F. J. B. Holljen. Compilation and type-safety in the Compose* .NET environment. Master's thesis, University of Twente, The Netherlands, May 2004.

[39] R. L. R. Huisman. Debugging Composition Filters. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[40] S. H. G. Huttenhuis. Patterns within aspect orientation. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[41] *Programming languages - C: ISO/IEC 9899:1999*. ISO/IEC, 1999.

[42] B. Kernigan and D. Ritchie. *The C programming Language*. Prentice Hall, 2 edition, 1988.

[43] 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, Berlin, June 2001. Springer-Verlag.

[44] 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.

[45] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, The Netherlands, June 2006.

[46] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[47] T. Parr, 2006. URL http://www.antlr.org.

[48] 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, Apr. 2002.

[49] A. Popovici, G. Alonso, and T. Gross. Just in time aspects. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, Mar. 2003.

[50] R. Pressman. *Software engineering a Practitioner's Approach*. McGraw-Hill Publishing Company, 2000. Adapted by D. Ince.

[51] M. Ségura-Devillechaise, J. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643615.

[52] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, Aug. 2001.

[53] R. Schmidt. Handling exceptions in c and c++, part 1, May 1999. URL http://msdn.microsoft.com/library/.

[54] C. Schwanninger, E. Wuchner, and M. Kircher. Encapsulating crosscutting concerns in system software. 2004. Proceedings of the Third AOSD Workshop on Aspects Components and Patterns for Infrastructure Software Held in conjunction with AOSD 2004 College of Computer and Information.

[55] D. R. Spenkelink. Compose* incremental. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[56] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7.

[57] T. Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, Apr. 2005.

[58] J. W. te Winkel. Bringing Composition Filters to C. Master's thesis, University of Twente, The Netherlands, 2006. To be released.

[59] M. D. W. van Oudheusden. Automatic derivation of semantic properties in .NET. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[60] C. Vinkes. Superimposition in the Composition Filters model. Master's thesis, University of Twente, The Netherlands, Oct. 2004.

[61] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[62] 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.

[63] C. Zhang and H. Jacobsen. Tinyc2:towards building a dynamic weaving aspect language for c. In *Foundations of Aspect-Oriented Languages 2003*, 2003.

Appendix A

Build Configuration

This appendix shows a buildconfiguration file that is used by Compose*. This file is automatically generated by the Eclipse-plugin.

```
<?xml version="1.0"?>
<BuildConfiguration version="1.00">
  <Projects applicationStart="EncryptionExampleTest.ExampleRunner"
  runDebugLevel="5" outputPath="H:\EncryptionExample\bin">
    <Project outputPath="H:\EncryptionExample\bin"
     buildPath="H:\EncryptionExample"
      basePath="H:\EncryptionExample" language="C" >
      <Sources>
        <Source fileName="H:\EncryptionExample\Server.c" />
        <Source fileName="H:\EncryptionExample\Encryptor.c" />
        <Source fileName="H:\EncryptionExample\Logger.c" />
        <Source fileName="H:\EncryptionExample\Protocol.c" />
        <Source fileName="H:\EncryptionExample\Client.c" />
        <Source fileName="H:\EncryptionExample\ExampleRunner.c"/>
       </Sources>
      <Dependencies>
      </Dependencies>
      <TypeSources>
      </TypeSources>
    </Project>
    <ConcernSources>
      <ConcernSource fileName=
        "H:\EncryptionExample\EncryptionConcern.cps" />
      <ConcernSource fileName=
        "H:\EncryptionExample\Logging.cps" />
     </ConcernSources>
  </Projects>
  <Settings composestarIni="C:/local/Johan/ComposeStar/Composestar.ini"
   buildDebugLevel="5" compilePhase="one" platform="c">
    <Modules>
      <Module name="INCRE" enabled="" />
      <Module name="SECRET" mode="" />
      <Module name="FILTH" input="" />
```

```
34        <Module name="ILICIT" verifyAssemblies="" />
35      </Modules>
36      <Paths>
37        <Path name="Output" pathName="H:\EncryptionExample\bin"/>
38        <Path name="Base" pathName="H:\EncryptionExample"/>
39        <Path name="Build" pathName="H:\EncryptionExample"/>
40        <Path name="Composestar" pathName=
41        "c:\Eclipse\plugins\CstarEP\"/>
42        <Path name="EmbeddedSources" pathName="embedded/" />
43      </Paths>
44    </Settings>
45    <Platforms>
46    <Platform name="C" mainClass="Composestar.C.MASTER.CMaster"
47      classPath=
48      "c:\Eclipse\plugins\CstarEP\binaries\ComposestarC.jar;
49      c:\Eclipse\plugins\CstarEP\binaries\ComposestarCORE.jar;
50      c:\Eclipse\plugins\CstarEP\binaries\antlr.jar;
51      c:\Eclipse\plugins\CstarEP\binaries\prolog.jar"
52      options="">
53      <Languages defaultLanguage="C">
54        <Language name="C">
55          <Compiler name="CCompiler" executable="gcc" options="/Wall"
56           implementedBy="Composestar.C.COMP.GCC">
57            <Actions>
58              <Action name="Compile" argument="{OPTIONS} {SOURCES}" />
59            </Actions>
60            <Converters />
61          </Compiler>
62          <FileExtensions>
63            <FileExtension extension=".c" />
64          </FileExtensions>
65      </Language>
66      </Languages>
67    </Platform>
68  </Platforms>
69  </BuildConfiguration>
```

Listing A.1: `BuildConfiguration` xml-file

Appendix B

Compose*C components

Compose⋆ C is an implementation of the language dependent part of Compose⋆. In this appendix the components that are contained by Compose⋆ C, are described briefly.

MASTER

Is an extension of the Compose⋆ Core component MASTER, and is responsible for initiating and running the different components in the right order.

WRAPPER

WRAPPER is the treeparser component coming from WeaveC. The original WRAPPER is extended for filling the repository with the right elements. WRAPPER has replaced TYM which is a component of Compose⋆ Core.

REXREF

The REXREF component is a small extension to the one in the Core. In Compose⋆ C REXREF also checks if there are internals or externals declared in the composition filters concerns that are specified in the CConcern.xml file. If this is the case REXREF creates concerns for these elements.

LAMA

In LAMA the C program elements are declared. These elements are instantiated from WRAPPER and added to the repository.

LOLA

LOLA describes the connections between the program elements of C. These connections can be used in the selector query. LOLA also evaluates the selector queries, and it creates a signature for the concerns defined in CConcern.xml.

CONE

The CONE component of Compose⋆ Core was used to generate a repository file and the weave instructions. In Compose⋆ C it is the component where the actual weaving is performed. It first generates the aspects from the composition filters concerns, and then executes the weaving.

SPECIFICATION

Is also a component originally belonging to WeaveC and contains the aspects and the belonging elements which are used for the weaving.

Appendix C

Creating a custom filter

A custom filter can be created by extending the Semantic class from Compose⋆ C. The Semantic class, described in Listing C.1, defines functions that offer concern information which can be used to specify source code, define the filter properties, and the source code that has to be woven.

```
1
2  public abstract class Semantic {
3
4       /**
5          * @return Code that creates the reified message struct
6          * for the matching function
7          */
8      protected String reifiedMessageCode();
9
10     /**
11         * @return The name of the condition function
12         */
13     protected String conditionCode();
14
15     /**
16         * @return ! if the operator used in the
17         * conditionexpression of the filter is ~>
18         */
19     protected String disableOperatorCode();
20
21     /**
22         * @return The name of the parameter of the matching function
23         * @param offset of the parameter from the function
24         */
25     protected String parameterName(int index);
26
27          /**
28         * @return The type of the annotation of
29         * the parameter of the matching function
30         * @param offset of the parameter from the function
```

```
31          */
32      protected CAnnotation parameterAnnotation(int index){
33
34           /**
35          * @return The annotation of a file
36          */
37      protected CAnnotation fileAnnotation();
38
39           /**
40          * @return The Annotation of a method
41          */
42      protected CAnnotation methodAnnotation();
43
44      /**
45        * @return The type of an annotation
46         */
47      protected String annotationType(CAnnotation anno);
48
49      /**
50        * @return Value of the annotation
51        */
52      protected String annotationValue(CAnnotation anno);
53
54      /**
55          * @return The type of the parameter of the matching function
56          * @param offset of the parameter from the function
57          */
58      protected String parameterTypeMF(int index);
59
60      /**
61          * @return The number of parameters of the matching function
62          *
63          */
64      protected int numberOfParametersMF();
65
66      /**
67          * @return The type of the parameter of the substitution function
68          * @param offset of the parameter from the function
69          */
70      protected String parameterTypeSF(int index);
71
72      /**
73          * @return The number of parameters of the substitution function
74          */
75      protected int numberOfParametersSF();
76
77      /**
78          * @return Name of the matching function
79          */
80      protected String matchingFunction();
81      /**
82          * @return Name of the substitution function
83          */
```

```
84      protected String substitutionFunction();
85
86          /**
87         * @return advice that has to be woven before the pointcut
88         */
89      public String getBeforeAdvice();
90          /**
91         * @return advice that has to be woven after the pointcut
92         */
93      public String getAfterAdvice();
94
95      /**
96         * @return returns the name of the filter
97         */
98      public abstract String getType();
99
100     /**
101        * @return  true if advice has to be woven after pointuct
102        */
103     public abstract boolean afterAdvice();
104
105     /**
106        * @return true if advice has to be woven before pointuct
107        */
108     public abstract boolean beforeAdvice();
109
110     /**
111        * @return true if the filter redirects a message to another
112        * function
113        */
114     public abstract boolean redirectMessage();
115
116     /**
117        * @return returns a name of a header file (including .h) as a
118        * string that contains a header where function or global
119        * definitions of elements used in the advice are specified
120        */
121     public abstract String headerFile();
122
123     /**
124        * @return true if there are other headers needed
125        * to make this filter work
126        */
127     public abstract boolean needsHeaderFiles();
128
129 }
```

Listing C.1: `Decrypt` custom filter

Creating a new custom filter requires to start a new Java project in Eclipse. The projectname needs to be the same as the filter name. The fully qualified name of the filter has to be used in the composition filters concern, therefore we propose not to define packages, only to create a class with the filter's name. This class is dependent of several

packages in Compose* C and Compose* Core, the easiest way to compile this is to add the jar files of these programs found in the ComposestarEclipsePlugin/binaries dir to the classpath. The variables in the Listing C.2 class define the properties of the filter. The decrypt filter only weaves before a pointcut, redirects a message to another function and does not use variables defined in another translation unit. The function getBeforeAdvice creates source code that shall be added before the pointcut.

```
import Composestar.C.CONE.Semantic;
public class Decryption extends Semantic{
    private String type="Decryption";
    private String advice="";
    private boolean afterAdvice=false;
    private boolean beforeAdvice=true;
    private boolean redirectMessage=false;
    private boolean needsHeaderFiles=false;

    /**Does not use parameter functions yet**/
    public String getBeforeAdvice(){
        String advice="";
        advice+= "char* x;";
        advice+= "x = "+ this.parameterName(0)+";";
        advice+= "printf(\"Decrypt message with filter:"
                          +this.parameterName(0)+" \n\");";
        advice+= "while(*x != \'\\0\'){";
        advice+= "*x -= 20;";
        advice+= "x++;";
        advice+= "}";
        return advice;
    }

    public String getAfterAdvice(){
        return advice;
    }

    public boolean afterAdvice(){
        return afterAdvice;
    }

    public boolean beforeAdvice(){
        return beforeAdvice;
    }

    public boolean redirectMessage(){
        return redirectMessage;
    }

    public String headerFile(){
        return "";
    }

    public boolean needsHeaderFiles(){
        return needsHeaderFiles;
    }
```

```
47
48      public String getType(){
49          return type;
50      }
51  }
```

Listing C.2: `Decrypt` custom filter

This class can be exported with eclipse to a jar-file. When the project is compiled with the Compose⋆-plugin, this jar-file can be selected. The plugin automatically adds this file to the classpath.

Appendix D

Installing Compose⋆ C and creating your first program

D.1 Installation

Using Compose⋆ C requires some specific configurations, these are:

Compose⋆ C
: Can be downloaded as a zip file. This zip file needs to be extracted in the plugin directory of the Eclipse installation;

Java
: Running Compose⋆ C requires an Java Runtime Environment 1.4 or higher, and its executables directory should be in the classpath;

GCC 2.95
: Just as Compose⋆ C the GCC version for Windows can be downloaded as a zip file. It does not bother where it is extracted, but the executables *gcc* and *make* should be in the classpath;

CDT-plugin
: Should be downloaded and installed. This is the standard C-plugin for Eclipse.

D.2 HelloWorld

When Compose⋆ C is installed a new application can be created. As an example we show here how to build HelloWorld with Compose⋆ C (the files are shown below). This requires the following steps:

1. Create a new standard make C project in Eclipse and name it HelloWorld;
2. Create the files HelloWorld.c and HelloWorld.h;

3. Create a concern for instance Lodewijk, with a .c .h .cps file;
4. Run it with Compose⋆ C from Eclipse. Here may occur some minor problems. When no output directory is created yet, Eclipse makes a mistake when you make it with the Compose⋆ C plugin. It says new folder in the input box instead of the folder name. This is an error of Eclipse self. Also the first time you run the project, a buildconfiguration file will be created, but it is not yet in the project. Therefore, the build process will return an error. Before running it again the project should be refreshed, than it will work.

```c
#include "HelloWorld.h"
int main(){
    printf("Hello World");
    worldsAnswer();
    return 0;
}
void worldsAnswer(){
    printf("Hello Pascal");
}
```

Listing D.1: `HelloWorld.c`

```c
#include <stdio.h>
int main();
void worldsAnswer();
```

Listing D.2: `HelloWorld.h`

```c
#include "Lodewijk.h"
void lodewijksAnswer()
{
    printf("Hello Lodewijk");
}
int isItLodewijk()
{
    return 1;
}
```

Listing D.3: `Lodewijk.c`

```c
#include <stdio.h>
int main();
void worldsAnswer();
```

Listing D.4: `Lodewijk.h` calcDivide

```
concern LodewijksAnswer
{
    filtermodule LAnswer
    {
        externals
            lodewijk : Lodewijk;
        conditions
```

```
8               isItLodewijk: lodewijk.isItLodewijk();
9           inputfilters
10              answer : Dispatch = {isItLodewijk=>
11                                  [*.worldsAnswer] *.lodewijksAnswer}
12      }
13
14      superimposition
15      {
16          selectors
17              HelloworldFile = { File | isFileWithName(File,'HelloWorld')};
18          filtermodules
19              HelloworldFile <- LAnswer;
20      }
21  }
```

Listing D.5: `Lodewijk.cps`

Appendix E

Demonstrating example

In this appendix we show the base code, applied concerns, and a part of the woven code, as well as the command-line output of the PolishCalculator. This example is used throughout this thesis.

Original program

```
#include "main.h"

int main(void)
{
    int type;
    double op2;
    char s[MAXOP];
    int flag = TRUE;
    double answer;

    printf("\nWelcome to the reverse polish calculator\n");
    printf("Calculating (1-2) * (4+5),
            requires an input of the form:\n");
    printf("1 2 - 4 5 + * (enter)\n");
    printf("The program can be terminated by using the letter: q \n");
    while((type = Getop(s)) != EOF)
    {
        switch(type)
        {
            case NUMBER:
                 ST_push(atof(s));
                break;
            case '+':
                if(calcAdd(&answer,ST_pop(),ST_pop()))
                    ST_push(answer);
                break;
            case '*':
```

```
28                  if(calcTimes(&answer,ST_pop(),ST_pop())==OK)
29                      ST_push(answer);
30                  break;
31              case '-':
32                  if(calcMinus(&answer,ST_pop(),ST_pop())==OK)
33                      ST_push(answer);
34                  break;
35              case '/':
36                  op2 = ST_pop();
37                  if(calcDivide(&answer, ST_pop(), op2)==OK)
38                      ST_push(answer);
39                  else
40                     printf("\nError: division by zero!");
41                  break;
42              case '%':
43                  op2 = ST_pop();
44                  if(calcModulo(&answer, ST_pop(), op2)==OK)
45                      ST_push(answer);
46                  else
47                     printf("\nError: modulo by zero!");
48                  break;
49              case '\n':
50                  printf("\n\t%.8g\n", ST_pop());
51                  break;
52              case 'q':
53                  printf("\nQuitted the polish calculator\n
54                                  ByBy please return soon\n");
55                  return 0;
56                  break;
57              default:
58                  printf("\nError: unknown command %s.\n", s);
59                  return -1;
60                  break;
61          }
62      }
63      return EXIT_SUCCESS;
64  }
```

Listing E.1: `Main.c`

```
1  #include<stdlib.h>
2  #include<stdio.h>
3  #include<ctype.h>
4  #include<math.h>
5
6  #define MAXOP 100
7  #define NUMBER  0
8  #define TRUE 1
9  #define FALSE 0
10 #define OK 2
11 #define ERROR 3
12
13 int Getop(char s[]);
14 void ST_push(double val);
```

```
15 double ST_pop(void);
```

Listing E.2: `Main.h`

```
1  #include <time.h>
2  #include "Calc.h"
3  struct annotatie{
4      char* x;
5  };
6  typedef struct annotatie in;
7  typedef struct annotatie out;
8  typedef struct annotatie inout;
9  int calcDivide($(inout("Semantic"))double* answer,
10     $(in("Semantic"))double op1, $(in("Semantic"))double op2)
11 {
12     int result=OK;
13     if(op2 != 0)
14         *answer = op1 / op2;
15     else
16         result =ERROR;
17     return result;
18 }
19 int calcMinus($(inout("Semantic"))double* answer,
20     $(in("Semantic"))double op1, $(in("Semantic"))double op2)
21 {
22     int result=OK;
23     *answer = op1 - op2;
24     return result;
25 }
26 int calcTimes($(inout("Semantic"))double* answer,
27     $(in("Semantic"))double op1, $(in("Semantic"))double op2)
28 {
29     int result=OK;
30     *answer = op1*op2;
31     return result;
32 }
33 int calcAdd($(inout("Semantic"))double* answer,
34     $(in("Semantic"))double op1, $(in("Semantic"))double op2)
35 {
36     int result=OK;
37     *answer = op1+op2;
38     return result;
39 }
40 int calcModulo($(inout("Semantic"))double* answer,
41     $(in("Semantic"))double op1, $(in("Semantic"))double op2)
42 {
43     int result=OK;
44     if(op2 != 0)
45         *answer = (int)op1%(int)op2;
46     else
47         result =ERROR;
48     return result;
```

```
}
```

Listing E.3: `Calc.c`

```
#include "main.h"
int calcDivide(double * answer , double op1 , double op2 );
int calcMinus(double * answer , double op1 , double op2 );
int calcAdd(double * answer , double op1 , double op2 );
int calcTimes(double * answer , double op1 , double op2 );
int calcModulo(double * answer , double op1 , double op2 );
```

Listing E.4: `Calc.h`

```
#include "main.h"
#define BUFSIZE 100
int getch(void);
void unGetch(int c);
char buf[BUFSIZE];
int bufp = 0;
/* Getop: get next operator or numeric operand. */
int Getop(char s[])
{
    int i = 0;
    int c;
    int next;
    /* Skip whitespace */
    while((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    /* Not a number but may contain a unary minus. */
    if(!isdigit(c) && c != '.' && c != '-')
        return c;
    if(c == '-')
    {
        next = getch();
        if(!isdigit(next) && next != '.')
        {
           return c;
        }
        c = next;
    }
    else
        c = getch();
    while(isdigit(s[++i] = c))
            c = getch();
    if(c == '.')                        /* Collect fraction part. */
        while(isdigit(s[++i] = c = getch()))
                        ;
    s[i] = '\0';
    if(c != EOF)
        unGetch(c);
    return NUMBER;
}
/* Getch: get a ( possibly pushed back) character. */
```

```
int getch(void)
{
     return (bufp > 0) ? buf[--bufp]: getchar();
}
/* unGetch: push character back on input. */
void unGetch(int c)
{
    if(bufp >= BUFSIZE)
        printf("\nUnGetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Listing E.5: `GetOperand.c`

```
#include "main.h"
#define MAXVAL 100
double val[MAXVAL];  /* value stack. */
int sp = 0;
/* push: push f onto stack. */
void ST_push(double f)
{
     val[sp++] = f;
}
/*pop: pop and return top value from stack.*/
double ST_pop(void)
{
    return val[--sp];
}
```

Listing E.6: `Stack.c`

Concerns

```
concern ErrorConcern {
    filtermodule error {
        externals
            error : Error;
            st: StackFunctions;
        conditions
            stackEmpty : error.stackEmpty();
            stackFull : error.stackFull();
        inputfilters
            error_filter : Error = { stackEmpty =>[*.ST_pop],
                stackFull =>[*.ST_push]}
    }
    superimposition {
        selectors
            stackFunctions = { C | isClassWithName(C, 'StackFunctions') };
        filtermodules
            stackFunctions <- error ;
    }
```

```
}
```

Listing E.7: `ErrorConcern.cps`

```
#include<stdio.h>
#include "Error.h"
extern int sp;
extern int bufp;
#define true 0;
#define false 1;
int stackFull(){
    if(sp==100){
        printf("Error: Stack full");
        return true;
    }
    else return false;
}
int stackEmpty(){
    if(sp==0){
        printf("Error: Stack empty");
        return true;
    }
    else return false;
}
int secondOpZero(){
    double op2;
    op2 = ST_pop();
        if(op2==0.0){
            printf("Error: second Operand zero");
            return false;
        }
}
int bufferFull(){
    if(bufp==100){
        printf("Error: Buffer full");
        return true;
    }
    return false;
}
```

Listing E.8: `Error.c`

```
#ifndef ERROR_H
#define ERROR_H
#include "main.h"
int stackFull();
int stackEmpty();
int secondOpZero();
#endif
```

Listing E.9: `Error.h`

```
concern ParameterCheckConcern {
 filtermodule Paramcheck {
```

```
    externals
        calc:CalculateFunctions;
    inputfilters
        paramcheck_filter : Parametercheck = { [calc.*] }
 }
 superimposition {
    selectors
        calcFunctions = { C | isClassWithName(C, 'CalculateFunctions') };
    filtermodules
        calcFunctions <- Paramcheck ;
 }
}
```

Listing E.10: ParameterCheck.cps

```
import Composestar.C.CONE.Semantic;
public class Parametercheck extends Semantic{
    private String type="Parametercheck";
    private String advice="";
    private boolean afterAdvice=false;
    private boolean beforeAdvice=true;
    private boolean redirectMessage=false;
    private boolean needsHeaderFiles=false;
    public String getBeforeAdvice(){
        String advice="";
        advice+= "printf(\"ParameterChecking Concern:"
            +this.matchingFunction()+" \");";
        for(int i=0; i<this.numberOfParametersMF(); i++){
            if(this.parameterAnnotation(i)!=null)
            {
                if(this.annotationType(parameterAnnotation(i)).
                    equals("in"))
                {
                    advice+= "if ("+this.parameterName(i)+ "== 0 ){";
                    advice+= "printf(\"Input parameter in function = NULL
                         function:"+this.matchingFunction()+"
                         parameterName= "+this.parameterName(i)
                         +" \\n\");";
                    advice+= "}";
                }
                if(this.annotationType(parameterAnnotation(i)).
                    equals("inout"))
                {
                    advice+= "if ("+this.parameterName(i)+ "== 0 ){";
                    advice+= "printf(\"InOut parameter in function = NULL
                        function:"+this.matchingFunction()+"
                        parameterName= "+this.parameterName(i)
                        +" \\n\");";
                    advice+= "}";
                }
                if(this.annotationType(parameterAnnotation(i)).
                    equals("out"))
                {
                    advice+= "if ("+this.parameterName(i)+ "== 0 ){";
```

```
                    advice+= "printf(\"Out parameter in function = NULL
                         function:"+this.matchingFunction()+
                         " parameterName= "+this.parameterName(i)
                            +" \\n\");";
                    advice+= "}";
                }
            }
        }
        return advice;
    }
    public String getAfterAdvice(){
        return advice;
    }
    public boolean afterAdvice(){
        return afterAdvice;
    }
    public boolean beforeAdvice(){
        return beforeAdvice;
    }
    public boolean redirectMessage(){
        return redirectMessage;
    }
    public String headerFile(){
        return "";
    }
    public boolean needsHeaderFiles(){
        return needsHeaderFiles;
    }
    public String getType(){
        return type;
    }
}
```

Listing E.11: `ParameterCheck.java`

```
concern Timing {
    filtermodule timing {
        inputfilters
            timing_filter : Timing = { [*.*] }
    }
    superimposition {
        selectors
            allFunctions = { File | isFileWithName(File,'Calc') };
        filtermodules
            allFunctions <- timing ;
    }
}
```

Listing E.12: `Timing.cps`

```
#include "Timing.h"
void TIMING_in(char* functionName, time_t time1)
{
    printf("Time 1 function: %s = %i\n", functionName, time1);
```

```
}
void TIMING_out(char* functionName, time_t time2){
    printf("Time 2 function: %s = %i\n", functionName, time2);
}
```

Listing E.13: Timing.c

```
import Composestar.C.CONE.Semantic;
public class Timing extends Semantic{
    private String type="Timing";
    private String advice="";
    private boolean afterAdvice=true;
    private boolean beforeAdvice=true;
    private boolean redirectMessage=false;
    private boolean needsHeaderFiles=true;
    public String getBeforeAdvice(){
        String advice="";
        advice+= "time_t time1;";
        advice+= "char * function_name = \""
                        + this.matchingFunction() +"\";";
        advice+= "(void)time(&time1);";
        advice+= "printf(\"\\n Timing Concern:
                        %s \\n\", function_name);";
        advice+="TIMING_in(function_name, time1);";
        return advice;
    }
    public String getAfterAdvice(){
        String advice= "time_t time2;";
        advice+="(void)time(&time2);";
        advice+= "printf(\"\\n Timing Concern:"
                    +this.matchingFunction()+" \\n\");";
        advice+="TIMING_out(\""
                    +this.matchingFunction()+"\", time2);";
        return advice;
    }
    public boolean afterAdvice(){
        return afterAdvice;
    }
    public boolean beforeAdvice(){
        return beforeAdvice;
    }
    public boolean redirectMessage(){
        return redirectMessage;
    }
    public String headerFile(){
        return "Timing.h";
    }
    public boolean needsHeaderFiles(){
        return needsHeaderFiles;
    }
    public String getType(){
        return type;
    }
```

```
}
```

Listing E.14: Timing.java

```
#ifndef TIMING_H
#define TIMING_H
#include <time.h>
#include <stdio.h>
void TIMING_in(char* functionName, time_t time1);
void TIMING_out(char* functionName, time_t time2);
#endif
```

Listing E.15: Timing.h

```
concern Tracing {
    filtermodule tracing {
        inputfilters
            tracing_filter : Tracing = { [*.*] }
    }
    superimposition {
        selectors
            allFunctions = { File | isFileWithName(File,'Calc') };
        filtermodules
            allFunctions <- tracing ;
    }
}
```

Listing E.16: Trace.cps

```
import Composestar.C.CONE.Semantic;
public class Tracing extends Semantic{
    private String type="Tracing";
    private String advice="";
    private boolean afterAdvice=true;
    private boolean beforeAdvice=true;
    private boolean redirectMessage=false;
    private boolean needsHeaderFiles=false;
    public String getBeforeAdvice(){
        String advice="";
        advice+= "printf(\"\\n Tracing Concern:"
                    +this.matchingFunction()+" \\n\");";
        advice+="TRACE_in(\""+this.matchingFunction()+"\");";
        return advice;
    }
    public String getAfterAdvice(){
        String advice="";
        advice+= "printf(\"\\n Tracing Concern:"
                    +this.matchingFunction()+" \\n\");";
        advice+="TRACE_out(\""+this.matchingFunction()+"\");";
        return advice;
    }
    public boolean afterAdvice(){
        return afterAdvice;
    }
```

```
26      public boolean beforeAdvice(){
27          return beforeAdvice;
28      }
29      public boolean redirectMessage(){
30          return redirectMessage;
31      }
32      public String headerFile(){
33          return "Trace.h";
34      }
35      public boolean needsHeaderFiles(){
36          return needsHeaderFiles;
37      }
38      public String getType(){
39          return type;
40      }
41  }
```

Listing E.17: `Tracing.java`

```
1   #include "Trace.h"
2   void TRACE_in(char* functionName)
3   {
4       printf("Trace In function: %s", functionName);
5   }
6   void TRACE_out(char* functionName){
7       printf("Trace Out function: %s", functionName);
8   }
```

Listing E.18: `Trace.c`

```
1   #ifndef TRACE_H
2   #define TRACE_H
3   #include <stdio.h>
4   void TRACE_in(char* functionName);
5   void TRACE_out(char* functionName);
6   #endif
```

Listing E.19: `Trace.h`

Result

```
1   struct annotatie {
2       char * x ;
3   } ;
4   typedef struct annotatie in ;
5   typedef struct annotatie out ;
6   typedef struct annotatie inout ;
7   #include "Error.h"
8   #include "message.h"
9   #include "Timing.h"
10  #include <assert.h>
11
12  int calcDivide ( double * answer , double op1 , double op2 )
```

```
{
    int result = 2 ;
#line 16 "Calc.c"

    {
        time_t time1 ; char * function_name = "calcDivide" ;
        ( void ) time ( & time1 ) ;
        printf ( "\n Timing Concern:%s \n" , function_name ) ;
        TIMING_in ( function_name , time1 ) ;
    }
    {
        printf ( "\n Tracing Concern:calcDivide \n" ) ;
        TRACE_in ( "calcDivide" ) ;
    }
    {
        printf ( "ParameterChecking Concern:calcDivide " ) ;
        if ( answer == 0 )
        {
            printf ( "InOut parameter in function =
                NULL function: calcDivide parameterName= answer \n");
        }
        if ( op1 == 0 )
        {
            printf ( "Input parameter in function = NULL function:
                calcDivide parameterName= op1 \n" ) ;
        }
        if ( op2 == 0 )
        {
            printf ( "Input parameter in function = NULL function:
                calcDivide parameterName= op2 \n" ) ;
        }
    }
    if ( op2 != 0 )
    * answer = op1 / op2 ;
    else
    result = 3 ;
#line 21 "Calc.c"
    {
        printf ( "\n Tracing Concern:calcDivide \n" ) ;
        TRACE_out ( "calcDivide" ) ;
        {
            time_t time2 ; ( void ) time ( & time2 ) ;
            printf ( "\n Timing Concern:calcDivide \n" ) ;
            TIMING_out ( "calcDivide" , time2 ) ;
            return result ;
        }
    }
}
```

Listing E.20: `Part of Calc.c after weaving`

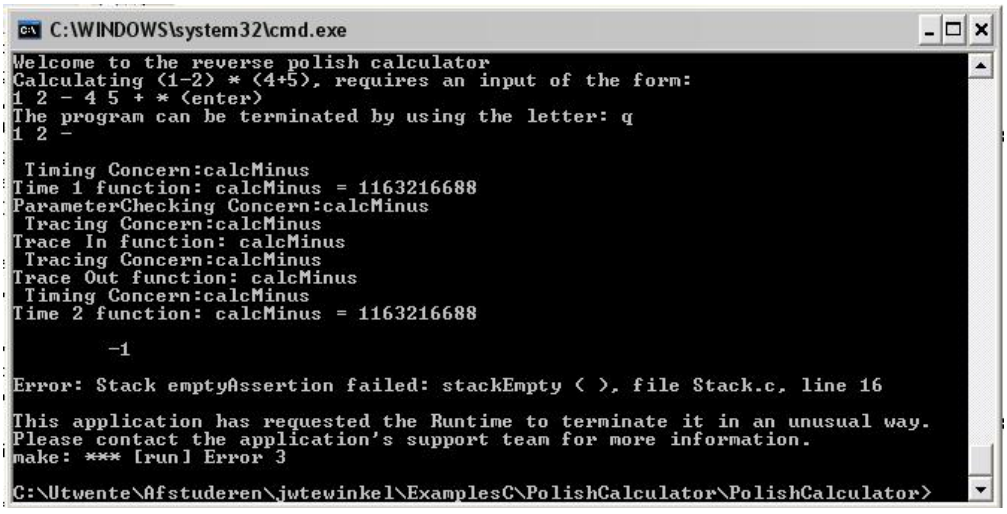


Figure E.1: PolishCalculator with concerns output

Appendix F

Eclipse-plugin

For usability reasons, Compose⋆ C is extended with an Eclipse-plugin. This plugin offers a user-interface, for compiling the project with Compose⋆ C and generates the build configuration-file as well as the makefile.

In Figure F.1 the added menu option in the Eclipse taskbar is seen. This menu has two implemented options, build and settings.

The build menu, Figure F.2, show the input-boxes mainclass, basepath, buildpath, outputpath, and custom filters. The mainclass is in principle not needed for Compose⋆ C, but is an option for Compose⋆ Java. The basepath is the position of the project, the build and outputpath, the position where it should be compiled, and where the executable should be placed. In the custom filter-box used filters can be selected.

The settings menu, Figure F.3, is only used for configuring the builddebuglevel and the classpath. The other options are for Compose⋆ Java.

The console window of Figure F.4 show the result of the EncryptionExample project which is build and run with Compose⋆ C from inside Eclipse.

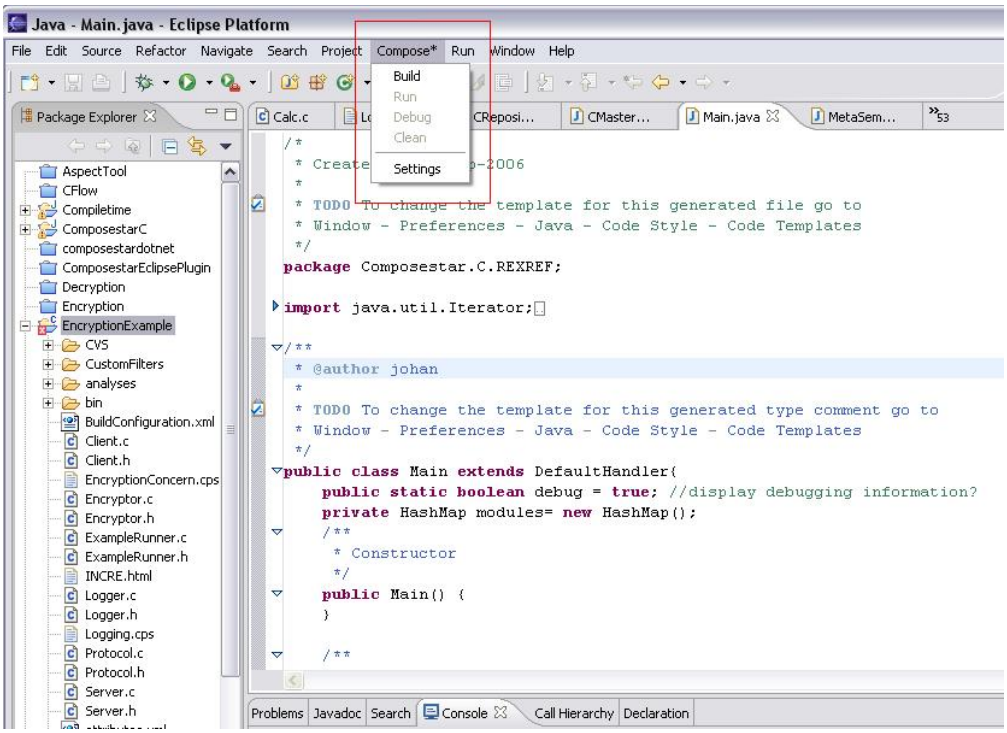


Figure F.1: Compose⋆ C menu in Eclipse

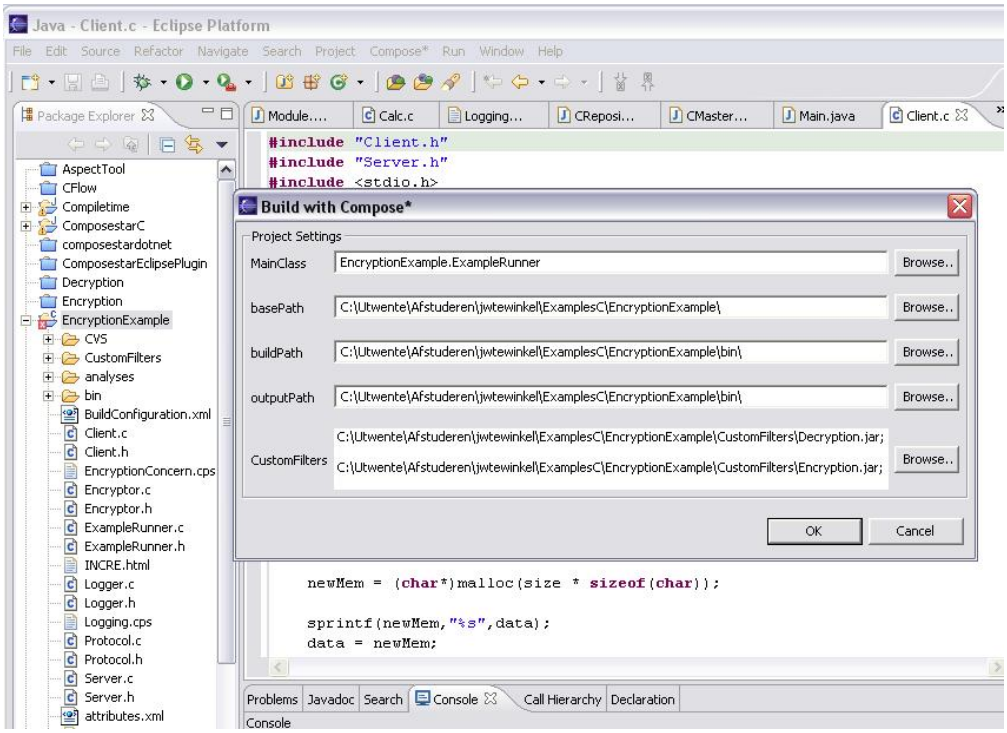


Figure F.2: Compose⋆ C build menu in Eclipse

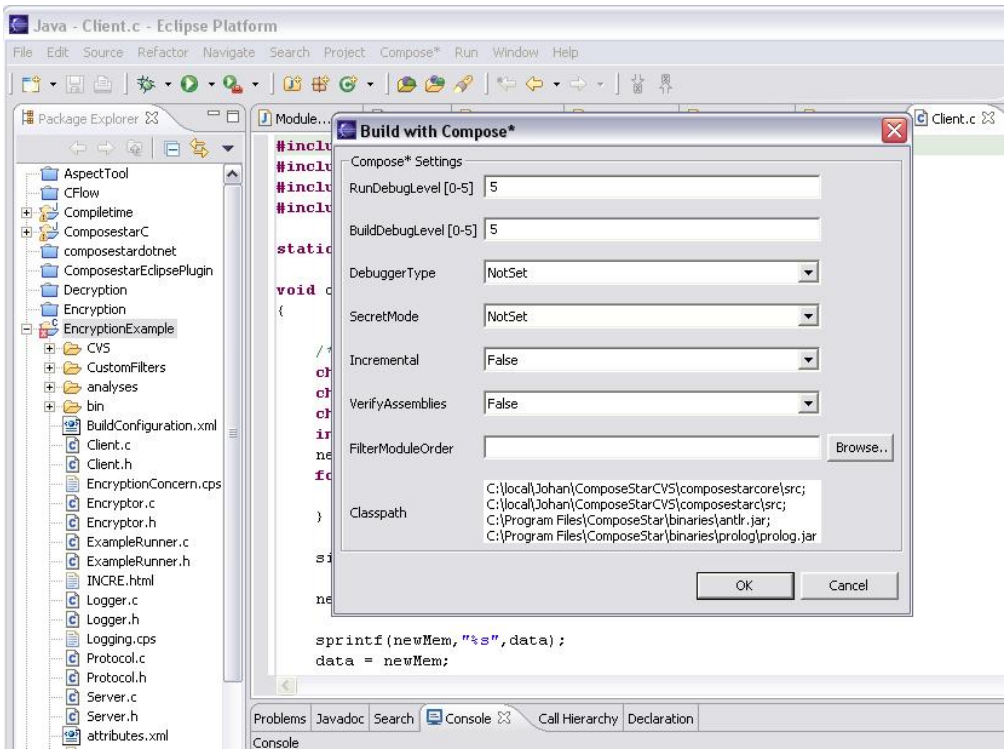


Figure F.3: Compose⋆ C settings menu in Eclipse

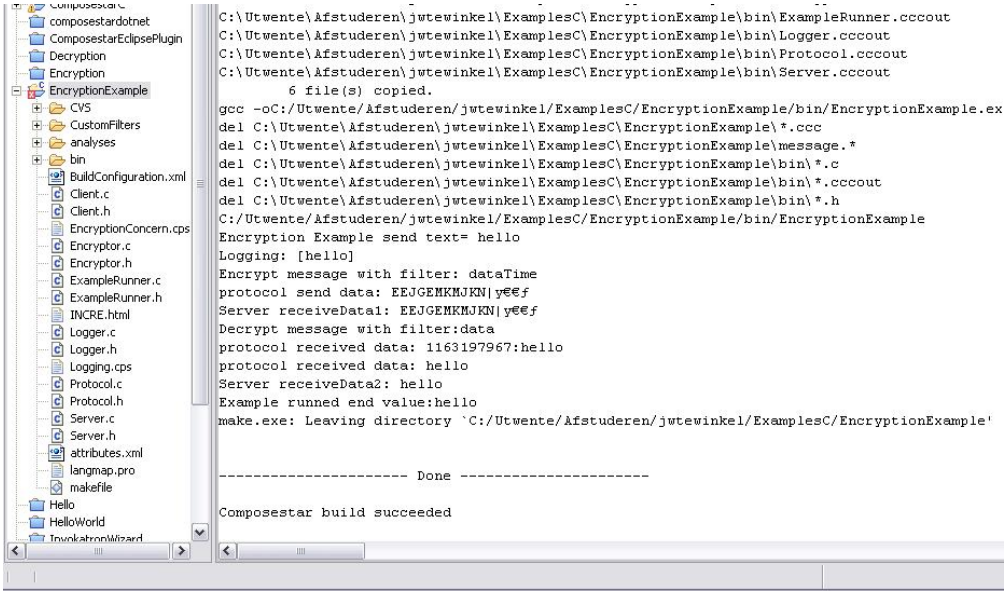


Figure F.4: Result of the EncryptionExample compiled with Compose⋆ C