Signature modification for Compose⋆/.NET

Implications of signature modification for generic compilers and behavioral signatures of types.

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

R. Jongeling

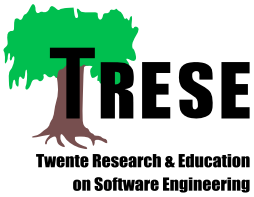
August 28, 2008

Research Group

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

Graduation Committee

dr. ir. L. M. J. Bergmans
prof. dr. ir. M. Akşit
ir. M. F. H. Hendriks



Abstract

Aspect-oriented programming solves one of the problems in object oriented programming by providing a better separation of concerns. The composition filter model is an implementation of AOP, which uses filtering of messages that are sent between objects. One of the techiques enabled by composition filters is signature modification. Signature modification allows composition filters to add methods to or remove methods from types.

This thesis consists of two parts. First, it presents a new solution to add support for signature modification to the compilation process of Compose⋆/.NET. Second, it presents an evaluation of extending the filter analysis part of Compose⋆ to include full behavioral analysis of the program.

The old solution that added support for signature modification to Compose⋆/.NET was found to be incompatible with the standard .NET 2.0 compilers. Furthermore, it is language dependent and lacks the support for self-referencing types. The current implementation of Compose⋆/.NET 2.0+ does not support signature modification.

A new solution is presented, that improves on the existing Compose⋆/.NET 2.0+ implementation. The improved compilation process introduces support for signature modification, is compatible with the .NET 2.0 compilers, is language independent and better maintainable than the previous solution. It does not add support for self-referencing types.

This thesis also presents an evaluation of extending the behavioral analysis performed by Compose⋆. The behavior of a type can be altered by the applied composition filters. The current behavioral analysis is limited to the behavior of composition filters. It describes this behavior with the resource-operation model. The behavioral analysis could be extended to include the behavior of the source code. This would enable Compose⋆ to detect composition filters that introduce conflicting behavior to a type.

The resource-operation model was found to be too generic and too abstract to be used to describe the behavior of a program. Extending the existing behavioral analysis to include analysis of source code is therefore difficult to achieve. An alternative is presented that extends the behavioral analysis to detect possible behavioral conflicts. This analysis is more limited than full behavioral analysis, but is easier to implement.

Acknowledgments

This thesis would not have been possible without the help of certain people, for which I am grateful.

I would like to thank Lodewijk Bergmans and Michiel Hendriks for their assistance and feedback. I also like to thank professor Mehmet Akşit. Without him, composition filters would not have existed in the first place.

Finally, I want to thank my family for all their patience and support.

Rolf Jongeling
August, 2008
Enschede, The Netherlands

Contents

List of Figures

List of Tables

Listings

Nomenclature

AOP	Aspect-Oriented Programming
API	Application Programming Interface
CF	Composition Filters
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
FCL	Framework Class Library
GUI	Graphical User Interface
IL	Intermediate Language
JIT	Just-in-time
JVM	Java Virtual Machine
OOP	Object-Oriented Programming
OpCode	Operation Code
PDA	Personal Digital Assistant
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
VM	Virtual Machine

WSDL	Web Services Description Language
XML	eXtensible Markup Language

Chapter 1

Introduction to AOSD

"Many people see AOSD as a solution.
Others see it as a research subject."
Rolf Huisman

The first two chapters have originally been written by seven M. Sc. students [4, 5, 10, 18, 20, 50, 57] at the University of Twente. The chapters have been rewritten for use in the following theses [6, 7, 9, 19, 21, 22, 25, 49, 53, 56]. They serve as a general introduction into Aspect-Oriented Software Development and Compose⋆ in particular.

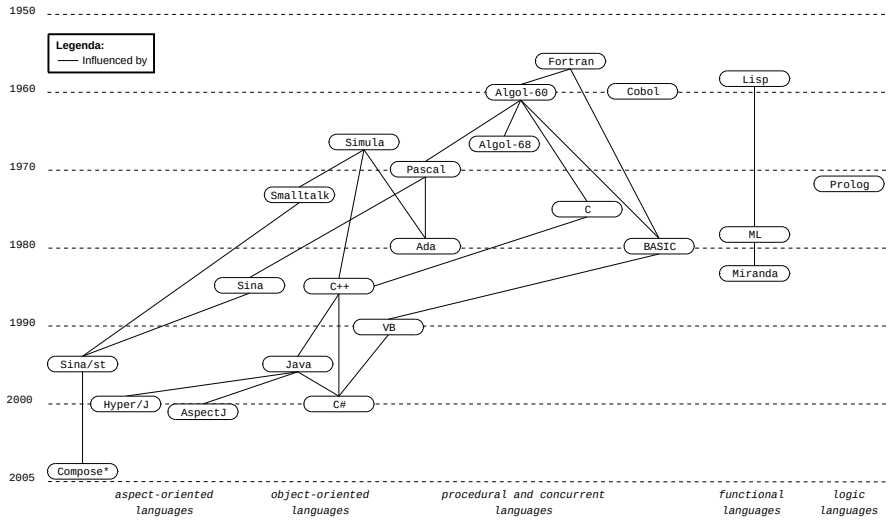


Figure 1.1: Dates and ancestry of several important languages

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 [59]. 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 [59].

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 [59]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The difficult part of 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 [14].

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 [52]. 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 one concern as 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 these. Finally, we look

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

(a) Addition

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

(b) CalcDisplay

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

at three particular AOP methodologies in more detail.

1.2 Traditional Approach

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

From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering*.

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the

other two classes (lines 5, 10, 14, and 20 in (a) and 2, 5, and 9 in (b)). 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, as are the display and tracing concerns. 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 `Add` or `CalcDisplay` object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

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

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being investigated 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 [16]: 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. The class `CalcDisplay` does not contain any tracing code either. 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.

```
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  public class CalcDisplay {
2    CalcDisplay() {
3    }
4
5    public void update(int value) {
6      System.out.println("Printing new
           value of calculation:
           "+value);
7    }
8  }
```

(b) CalcDisplay concern

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

(c) Tracing concern

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

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 e.g. Hyper/J.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. This approach is followed by e.g. AspectJ (covered in more detail in the next section).

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. There are three approaches to aspect weaving. The first and second approach rely on adding behavior to the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be an 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 compiler-type).

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. Then, the source code is 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 identified in Section 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that cannot 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
It is possible to add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or runtime
A special class loader or runtime environment can do dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend on the implementation of the runtime 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 the disadvantages of intermediate language weaving, mentioned in Section 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [45, 46].

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 spent 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 [13] 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, the possiblity to monitor performance and the possiblity to debug the aspects.

This section will give a short introduction to AspectJ [27] and Hyperspaces [43], which together with Composition Filters [3] are three main AOP approaches.

1.4.1 AspectJ Approach

AspectJ [27] 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

```
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 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, built 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`

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

Listing 1.4: Example of static crosscutting in AspectJ

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, which specifies where the additional code is to be inserted. In the example both before and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

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

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (super classes 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 separation of concerns [43], 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, though 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 byte code 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

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

Listing 1.5: Creation of a hyperspace

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, 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.

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

Listing 1.6: Specification of concern mappings

```
1 hypermodule Pacman_Without_Debugging
2   hyperslices: Feature.Kernel, Feature.Logging;
3   relationships: mergeByName;
4 end hypermodule;
```

Listing 1.7: Defining a hypermodule

1.4.3 Composition Filters

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

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

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, thus 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 sent message 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 need to be superimposed on which inner objects.

Chapter 2

Introduction to the .NET Framework

"The best way to prepare [to be a programmer] is to write programs,
and to study great programs that other people have written.
In my case, I went to the garbage cans at the Computer Science Center
and fished out listings of their operating system."
William Henry Gates III

This chapter gives an introduction to the .NET Framework of Microsoft. First, the architecture of the .NET Framework is introduced. This section includes terms like the Common Language Runtime, the .NET Class Library, the Common Language Infrastructure and the Intermediate Language. These are discussed in more detail in the sections following the architecture.

2.1 Introduction

Microsoft defines .NET as follows; ".NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software." [35]. There are different .NET technologies in various Microsoft products providing the capabilities to create solutions using web services. Web services are small, reusable applications that help computers from many different operating system platforms to work together by exchanging messages. Based on industry standards like XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language), they provide a platform and language independent way to communicate.

Microsoft products, such as Windows Server System (providing web services) or Office System (using web services) are some of the .NET technologies. The technology described in this chapter is the .NET Framework. Together with Visual Studio, an integrated development environment, they provide the developer with tools to create programs for .NET.

Many companies are largely dependent on the .NET Framework, but need or want to use AOP. Currently, there is no direct support for this in the Framework. The

Compose★/.NET project is addressing these needs with its implementation of the Composition Filters approach for the .NET Framework.

This specific Compose★ version for .NET has two main goals. First, it combines the .NET Framework with AOP through Composition Filters. Second, Compose★ offers superimposition in a language independent manner. The .NET Framework supports multiple languages and is, as such, suitable for this purpose. Composition Filters are an extension of the object-oriented mechanism as offered by .NET, hence the implementation is not restricted to any specific object-oriented language.

2.2 Architecture of the .NET Framework

The .NET Framework is Microsoft's platform for building, deploying, and running Web Services and applications. It is designed from scratch and has a consistent API providing support for component-based programs and Internet programming. This new Application Programming Interface (API) has become an integral component of Windows. The .NET Framework was designed to fulfill the following objectives [32]:

Consistency
: Allow object code to be stored and executed locally, executed locally but Internet-distributed, or executed remotely and to make the developer experience consistent across a wide variety of types of applications, such as Windows-based applications and Web-based applications;

Operability
: The ease of operation is enhanced by minimizing version conflicts and providing better software deployment support;

Security
: All the code is executed safely, including code created by an unknown or semi-trusted third party;

Efficiency
: The .NET Framework compiles applications to machine code before running, thus eliminating the performance problems of scripted or interpreted environments;

Interoperability
: Code based on the .NET Framework can integrate with other code because all communication is built on industry standards.

The .NET Framework consists of two main components [32]: the Common Language Runtime (CLR, simply called the .NET Runtime or Runtime for short) and the .NET Framework Class Library (FCL). The CLR is the foundation of the .NET Framework, executing the code and providing the core services such as memory management, thread management and exception handling. The CLR is described in more detail in Section 2.3. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to

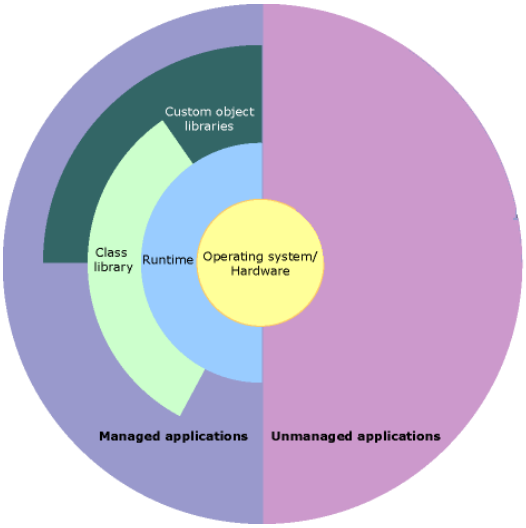


Figure 2.1: Context of the .NET Framework (Modified) [32]

applications such as Web Forms and XML Web services. Section 2.5 describes the class libraries in more detail.

The code run by the runtime is in a format called the Common Intermediate Language (CIL), further explained in Section 2.6. The Common Language Infrastructure (CLI) is an open specification that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. Section 2.4 tells more about this specification.

Figure 2.1 shows the relationship of the .NET Framework to other applications and to the complete system. The two parts, the class library and the runtime, are managed, i .e., applications managed during execution. The operating system is in the core, managed and unmanaged applications operate on the hardware. The runtime can use other object libraries and the class library, but the other libraries can use the same class library themselves.

Besides the Framework, Microsoft also provides a developer tool called the Visual Studio. This is an IDE with functionality across a wide range of areas, allowing developers to build applications with decreased development time in comparison with developing applications using command line compilers.

2.2.1 Version 2.0 of .NET

In November 2005, Microsoft released a successor of the .NET Framework. Major changes are the support for generics, the addition of nullable types, 64 bit support, improvements in the garbage collector, new security features and more network functionality.

Generics make it possible to declare and define classes, structures, interfaces, methods and delegates with unspecified or generic type parameters instead of specific types.

When the generic is used, the actual type is specified. This allows for type-safety at compile-time. Without generics, the use of casting or boxing and unboxing decreases performance. By using a generic type, the risks and costs of these operations is reduced.

Nullable types allow a value type to have a normal value or a null value. This null value can be useful for indicating that a variable has no defined value because the information is not currently available.

Besides changes in the Framework, there are also improvements in the four main Microsoft .NET programming languages (C#, VB.NET, J# and C++). The language elements are now almost equal for all languages. For instance, additions to the Visual Basic language are the support for unsigned values and new operators. Additions to the C# language include the ability to define anonymous methods, thus eliminating the need to create a separate method.

A new edition of Visual Studio (Visual Studio 2005) was released to support the new Framework and functionalities to create various types of applications.

2.2.2 Version 3.0 and 3.5 of .NET

Version 3.0 of the .NET framework was released together with Windows Vista. It uses the Common Language Runtime of the .NET 2.0 framework. The .NET 3.0 framework introduces four new major components, which mostly add support for Windows Vista and Windows Server 2008.

The *Windows Presentation Foundation* (WPF) is a new UI subsystem and API. The *Windows Communication Foundation* (WCF) is a new service-oriented messaging system. The *Windows Workflow Foundation* (WF) allows using workflows to build taks automation and integrate transactions. The *Windows CardSpace* (WCS) provides secure storage and management of a person's digital identity.

Version 3.5 was released in November 2007. It is still based on the Common Language Runtime of .NET 2.0, but a number of improvements and additions have been made (most notably in cryptography and networking).

The biggest new feature in .NET 3.5 is the *Language-Integrated Query* (LINQ). LINQ extends query capabilities to the language syntax of C# and VB.NET in the form of standard query patterns. This technology can be extended to support potentially any kind of data store; for example, LINQ support is provided for XML and SQL data.

Version 3.5 introduces new versions of the C# and VB.NET programming languages. C# 3.0 introduces new language constructions, such as anonymous types, lambda expressions and support for LINQ. C# 3.0 is designed to be binary-compabible with C# 2.0.

Visual Studio 2008 was released with .NET 3.5. It introduces improved support for features introduced in .NET 3.0 and .NET 3.5, including support for C# 3.0, VB.NET 9, LINQ and WPF. However, the Visual J# programming language is no longer included with Visual Studio 2008.

2.2.3 Mono

The Mono project was started in 2001 as an open source, cross-platform development platform based on the .NET framework. It is based on the ECMA spefications for C# and the CLI. Currently, it supports a number of Windows and Linux platforms.

Development tools Mono offers tools for development with the Mono framework. These tools include an IDE, debugging tools and a documentation browser. Mono also offers a C# compiler, which is designed to create .NET assemblies that can run on both the Mono and Microsoft's .NET frameworks.

ECMA-compatible runtime (CLR) Mono's common language runtime is designed to run all .NET assemblies, including assemblies developed for and compiled by the Microsoft .NET framework.

Libraries The Mono framework includes the base class library for compatibility with Microsoft, the Mono class library to provide additional functionality and third-party libraries (including Gtk#, wich provides Gtk+ support for .NET).

Version 1.0 of the Mono framework was released June 2004 and implements the .NET 1.1 API. The release includes a C# compiler, a virtual machine with JIT, an IL assembler and disassembler and an implementation of the .NET 1.1 libraries.

The release of Mono 2.0 is slated for September 2008 and is designed to be compatible with the .NET 2.0 API. The release will include a compiler for the C# 3.0 language (including support for LINQ), a VB.NET 8 compiler and runtime, new delopment tools and libraries for .NET 2.0 and additional fuctionality. Futhermore, the Mac OS X platform will be supported.

2.3 Common Language Runtime

The Common Language Runtime executes code and provides core services. These core services are memory management, thread execution, code safety verification and compilation. Apart from providing services, the CLR also enforces code access security and code robustness. Code access security is enforced by providing varying degrees of trust to components based on a number of factors, e .g., the origin of a component. This way, a managed component might or might not be able to perform sensitive functions, like file-access or registry-access. By implementing a strict type-and-code-verification infrastructure, called the Common Type System (CTS), the CLR enforces code robustness. Basically there are two types of code;

Managed

Managed code is code that has its memory handled and its types validated at execution by the CLR. It has to conform to the Common Type Specification (CTS Section 2.4). If interoperability with components written in other languages is required, managed code has to conform to an even more strict set of specifications,

the Common Language Specification (CLS). The code is run by the CLR and is typically stored in an intermediate language format. This platform independent intermediate language is officially known as Common Intermediate Language (CIL) (see Section 2.6) [58].

Unmanaged

Unmanaged code is not managed by the CLR. It is stored in the native machine language and is not run by the runtime but directly by the processor.

All language compilers targeting the CLR, generate managed code (CIL) that conforms to the CTS.

At runtime, the CLR is responsible for generating platform specific code, which can actually be executed on the target platform. Compiling from CIL to the native machine language of the platform is executed by the just-in-time (JIT) compiler. Because of this language independent layer it allows the development of CLRs for any platform, creating a true interoperability infrastructure [58]. The .NET Runtime from Microsoft is actually a specific CLR implementation for the Windows platform. Microsoft has released the *.NET Compact Framework* especially for devices such as personal digital assistants (PDAs) and mobile phones. The .NET Compact Framework contains a subset of the normal .NET Framework and allows .NET developer to write mobile applications. Components can be exchanged and web services can be used so that an easier interoperability between mobile devices and workstations/servers can be implemented [34].

The .NET and the Mono Framework are the only advanced Common Language Infrastructure (CLI) implementations available, but a number of other implementations are being developed. A shared-source[1] implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [51]. In 2006 Microsoft released an updated version of Rotor for the .NET platform version two. Another, somewhat different approach, is called Plataforma.NET[2]. Plataforma.NET aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

2.3.1 Java VM vs .NET CLR

There are many similarities between Java and .NET technology. This is not strange, because both products serve the same market.

Both Java and .NET are based on a runtime environment and an extensive development framework. These development frameworks provide largely the same functionality for both Java and .NET. The most obvious difference between them is lack of language independence in Java. While Java's strategy is 'One language for all platforms', the .NET philosophy is 'All languages on one platform'. However these philosophies are not as strict as they seem. As noted in Section 2.5 (and shown by Mono), there is no technical obstacle for other platforms to implement the .NET Framework. There are compilers for non-Java languages like Jython (Python) [26] and WebADA [1] available for the JVM. Thus, the JVM in its current state has difficulties supporting such a vast

[1] Only non-commercial purposes are allowed.

[2] `http://personals.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html`

array of languages as the CLR. However, the multiple language support in .NET is not optimal and has been the target of some criticism.

Although the JVM and the CLR provide the same basic features, they differ in some ways. While both CLR and the modern JVM use JIT (Just In Time) compilation, the CLR can directly access native functions. This means that with the JVM an indirect mapping is needed to interface directly with the operating system.

2.4 Common Language Infrastructure

The entire CLI has been documented, standardized and approved [23] by the European association for standardizing information and communication systems, Ecma International[1]. This CLI provides a number of benefits for developers and end-users:

- Most high level programming languages can easily be mapped onto the Common Type System (CTS);
- The same application will run on different CLI implementations;
- Cross-programming language integration, if the code strictly conforms to the Common Language Specification (CLS);
- Different CLI implementations can communicate with each other, providing applications with easy cross-platform communication means.

This interoperability and portability is, for instance, achieved by using a standardized meta data and intermediate language (CIL) scheme as the storage and distribution format for applications. In other words, (almost) any programming language can be mapped to CIL, which in turn can be mapped to any native machine language.

The Common Language Specification is a subset of the Common Type System, and defines the basic set of language features that all .NET languages should adhere to. In this way, the CLS helps to enhance and ensure language interoperability by defining a set of features that are available in a wide variety of languages. The CLS was designed to include all the language constructs that are commonly needed by developers (e.g., naming conventions, common primitive types), but no more than most languages are able to support [33]. Figure 2.2 shows the relationships between the CTS, the CLS, and the types available in C++ and C#. In this way the standardized CLI provides, in theory[2], a true cross-language and cross-platform development and runtime environment.

To attract a large number of developers for the .NET Framework, Microsoft has released CIL compilers for C++, C#, J#, and VB.NET. In addition, third-party vendors and open-source projects also released compilers targeting the .NET Framework,

[1] An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at `http://www.ecma-international.org/`.

[2] Unfortunately Microsoft did not submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.

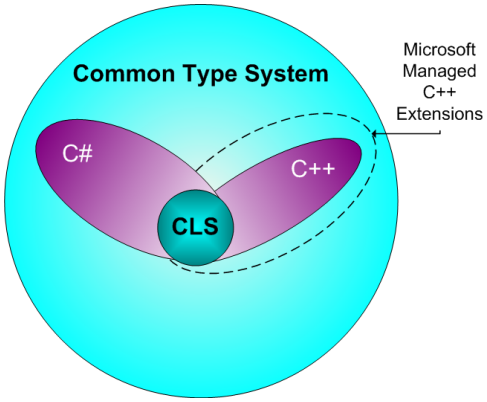


Figure 2.2: Relationships in the CTS

such as Delphi.NET, Perl.NET, IronPython, and Eiffel.NET. These programming languages cover a wide-range of different programming paradigms, such as classic imperative, object-oriented, scripting, and declarative languages. This wide coverage demonstrates the power of the standardized CLI.

Figure 2.3 shows the relationships between all the main components of the CLI. The top of the figure shows the different programming languages with compiler support for the CLI. Because the compiled code is stored and distributed in the Common Intermediate Language format, the code can run on any CLR. For cross-language usage this code has to comply with the CLS. Any application can use the class library (the FCL) for common and specialized programming tasks.

2.5 Framework Class Library

The .NET Framework class library is a comprehensive collection of object-oriented reusable types for the CLR. This library is the foundation on which all the .NET applications are built. It is object oriented and provides integration of third-party components with the classes in the .NET Framework. Developers can use components provided by the .NET Framework, other developers and their own components. A wide range of common programming tasks (e .g., string management, data collection, reflection, graphics, database connectivity or file access) can be accomplished easily by using the class library. Also, a great number of specialized development tasks are extensively supported, like:

- Console applications;
- Windows GUI applications (Windows Forms);
- Web applications (Web Forms);
- XML Web services;
- Windows services.

All the types in this framework are CLS compliant and can therefore be used from any programming language whose compiler conforms to the Common Language Specification

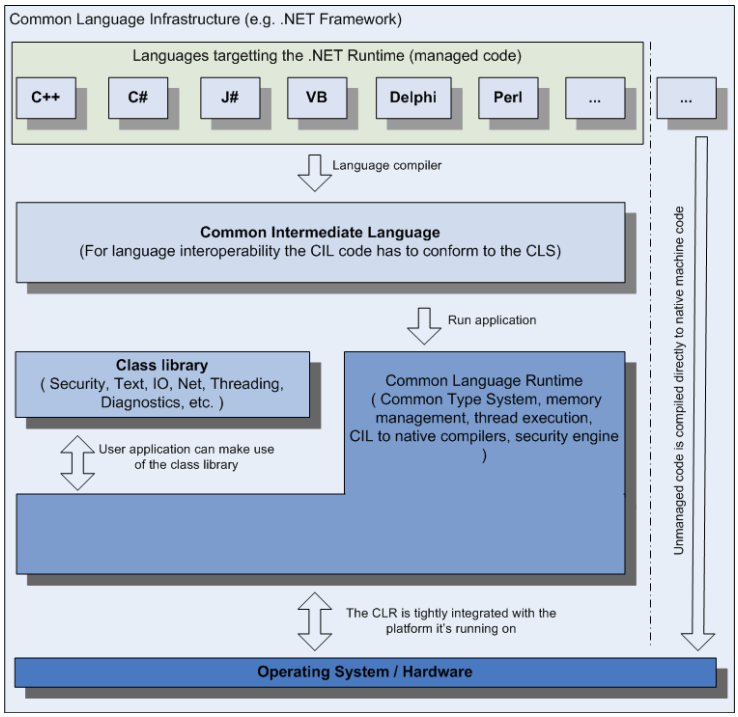


Figure 2.3: Main components of the CLI and their relationships. The right hand side of the figure shows the difference between managed code and unmanaged code.

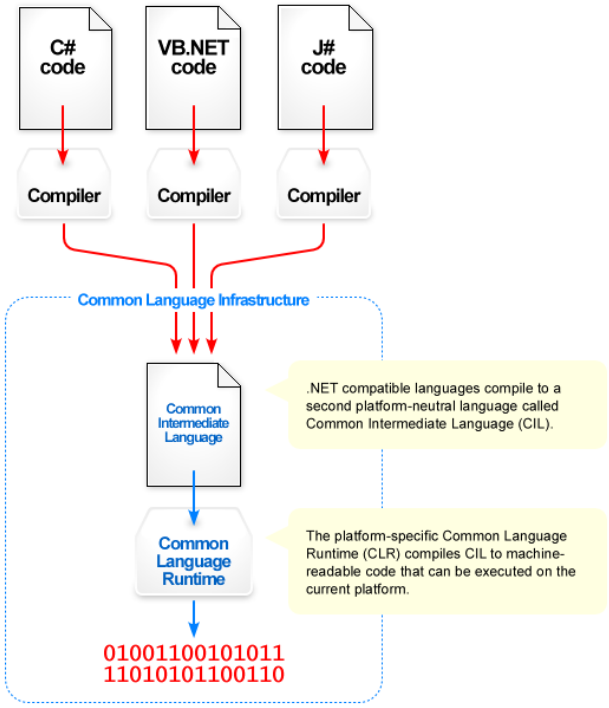


Figure 2.4: From source code to machine code

(CLS).

2.6 Common Intermediate Language

The Common Intermediate Language (CIL) has already been briefly mentioned in the sections before, but this section will describe the CIL in more detail. All the languages targeting the .NET Framework compile to this CIL (see Figure 2.4).

A .NET compiler generates a *managed module*, which is an executable designed to be run by the CLR [47]. There are four main elements inside a managed module:

- A Windows Portable Executable (PE) file header;
- A CLR header containing important information about the module, such as the location of its CIL and metadata;
- Metadata describing everything inside the module and its external dependencies;
- The CIL instructions generated from the source code.

The Portable Executable file header allows the user to start the executable. This small piece of code will initiate the just-in-time compiler which compiles the CIL instructions to native code when needed, while using the metadata for extra information about the program. This native code is machine dependent while the original IL code is still

machine independent. In this way, the same IL code can be JIT-compiled and executed on any supported architecture. The CLR cannot use the managed module directly but needs an assembly.

An assembly is the fundamental unit of security, versioning, and deployment in the .NET Framework. It is a collection of one or more modules and/or files grouped together to form a logical unit [47]. Besides managed modules inside an assembly, it is also possible to include resources like images or text. A manifest file is contained in the assembly, describing not only the name, culture and version of the assembly but also the references to other files in the assembly and security requests.

The CIL is an object oriented assembly language with around 100 different instructions called OpCodes. It is stack-based, meaning objects are placed on an evaluation stack before the execution of an operation, and when applicable, the result can be found on the stack after the operation. For instance, if two numbers have to be added, first those numbers are placed onto the stack, then the add operation is called and finally the result can be retrieved from the stack.

```
1  .assembly AddExample {}
2
3  .method static public void main() il managed
4  {
5    .entrypoint            // entry point of the application
6    .maxstack 2
7
8    ldc.i4 3               // Place a 32-bit (i4) 3 onto the stack
9    ldc.i4 7               // Place a 32-bit (i4) 7 onto the stack
10
11   add                    // Add the two and
12                          // leave the sum on the stack
13
14   // Call static System.Console.Writeline function
15   // (function pops integer from the stack)
16   call void [mscorlib]System.Console::WriteLine(int32)
17
18   ret
19 }
```

Listing 2.1: Adding example in IL code

To illustrate how to create a .NET program in IL code we use the previous example of adding two numbers and show the result. In Listing 2.1 a new assembly is created with the name `AddExample`. In this assembly, a function `main` is declared as the starting point (`entrypoint`) of the assembly. The `maxstack` command indicates there can be a maximum of two objects on the stack. This is enough for the example method. Next, the values 3 and 7 are placed onto the stack. The `add` operation is called and the result stays on the stack. The method `WriteLine` from the .NET Framework Class Library is called. This method resides inside the `Console` class placed in the `System` assembly. It expects one parameter with `int32` as its type. This parameter will be retrieved from the stack. The `call` operation will transfer the control flow to this method, passing along the parameters as objects on the stack. The `WriteLine` method does not return a value.

The `ret` operation returns the control flow from the main method to the calling method, in this case the runtime. This will exit the program.

To be able to run this example, we need to compile the IL code to byte code where each OpCode is represented as one byte. To compile this example, save it as a text file and run the *ILASM* compiler with the filename as parameter. This will produce an executable that is runnable on all the platforms where the .NET Framework is installed.

This example was written directly in IL code, but we could have used a higher level language such as C# or VB.NET. For instance, the same example in C# code is shown in Listing 2.2 and the VB.NET version is listed in Listing 2.3. When this code is compiled to IL, it will look like the code in Listing 2.1.

```
1 public static void main()
2 {
3       Console.WriteLine((int) (3 + 7));
4 }
```

Listing 2.2: Adding example in the C# language

```
1 Public Shared Sub main()
2       Console.WriteLine(CType((3 + 7), Integer))
3 End Sub
```

Listing 2.3: Adding example in the VB.NET language

Chapter 3

Compose⋆

"The difficult part of composition filters
is understanding its simplicity."
Lodewijk Bergmans

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

3.1 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 [28].

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. 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 [2].

1995 The Sina language with Composition Filters is implemented using Smalltalk [28]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [15].

1999 The Composition Filters language ComposeJ [60] 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 [48]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

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

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

2005 The start of the Java port of Compose⋆.

2006 Porting Compose⋆ to C is started.

2006 Start of the StarLight project. This project is described in detail in Section 3.7.

2007 Merged the StarLight branch with Compose⋆.

3.2 Composition Filters in Compose⋆

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

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 3.1: Abstract concern template

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 to and from superimposed objects. Messages have a target, which is an object reference, and a selector, which is a method name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 3.1.

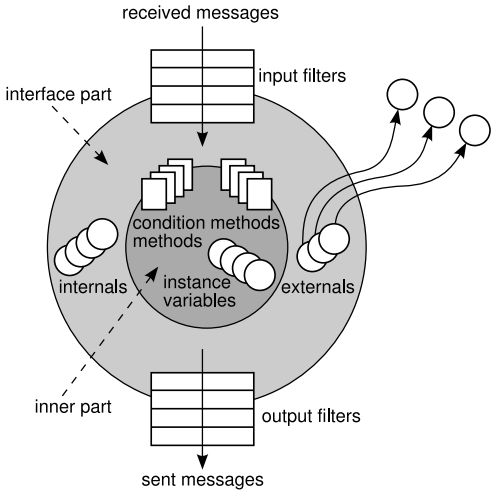


Figure 3.1: Components of the composition filters model

The working of a filter module is depicted in Figure 3.1. 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. If the condition part and the matching part are true, the message is substituted with the values provided in the substitution part. How these values are substituted, and how the message continues, depends on the type of filter used.

At the moment, there are four basic filter types defined in Compose⋆. However, it is 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.

3.3 Demonstrating Example

To illustrate the Compose⋆ toolset, this section introduces a *Pac-Man* example. The Pac-Man game is a classic arcade game in which the user, represented by Pac-Man, moves in a maze to eat pills. Meanwhile, a number of ghosts try to catch and eat Pac-Man. There are, however, four power pills in the maze that make Pac-Man evil. In its evil state, Pac-Man can eat the ghosts.

A simple list of requirements for the Pac-Man game is briefly discussed here:

- The number of lives taken from Pac-Man when eaten by a ghost;
- A game should end when Pac-Man has no more lives;
- The score of a game should increase when Pac-Man eats a pill or a ghost;
- A user should be able to use a keyboard to move Pac-Man around the maze;
- Ghosts should know whether Pac-Man is evil or not;
- Ghosts should know where Pac-Man is located;
- Ghosts should hunt or flee from Pac-Man, depending on the state of Pac-Man;
- At fixed intervals a bonus item should appear that gives bonus points when eaten by Pac-Man.

3.3.1 Initial Object-Oriented Design

Figure 3.2 shows an initial object-oriented design for the Pac-Man game. Note that this UML class diagram does not show the class attributes. The classes in this diagram are:

AIController The `AIController` is used for the non user controlled pawns. It makes use of a `Strategy` class to determine the next move;

Controller The controller is responsible for providing the `Pawn` with the desired direction of movement. A controller is either user controller or computer controlled;

Game This class encapsulates the control flow and controls the state of a game;

GameElement This is the base class for all, with the exception of the pills, visual elements in the game. A `GameElement` has a position within the maze and a collision radius;

Ghost This class is a representation of a ghost chasing Pac-Man. Each ghost has an identifier that can be used to change the behavior for individual ghosts;

HumanController The `HumanController` is an abstract controller class that receives movement instructions from a user;

KeyboardController The `KeyboardController` is an implementation of the `HumanController` that reads the instructions from the keyboard;

Level This class contains the structural information of the maze, this includes information about the walls, location of the pills, and starting locations of Pac-Man and the ghosts. It contains methods to check

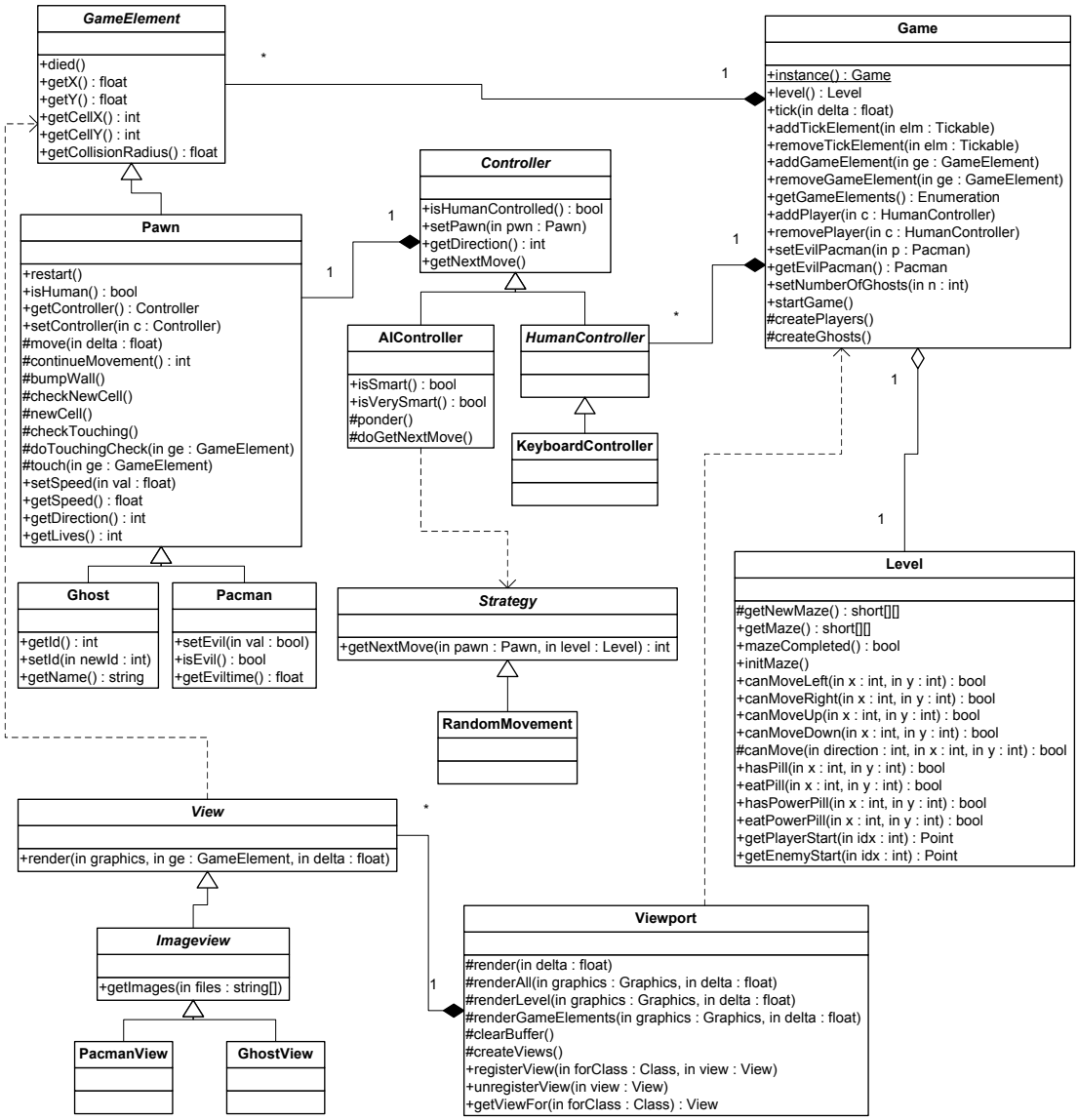


Figure 3.2: Class diagram of the object-oriented Pac-Man game

for, and eat the pills in the maze. It also has method to check for walls in a given direction;

Pacman This is a representation of the user controlled pawn in the game. It contains a flag that tells if the Pac-Man is in evil mode;

Pawn The `Pawn` class is used for all non stationary `GameElements`. Pawns are controlled by an instance of the `Controller` class. It has a speed, direction and number of lives. The pawn performs collision checks after every move and acts accordingly. A pawn will continue moving in the set direction with the given speed until these parameters have been changed;

RandomMovement The `RandomMovement` class returns a random direction the given pawn can move to;

Strategy This class is used by the `AIController` to make a movement choice;

View `GameElements` are drawn on the `Viewport` using a `View` implementation that has been registered with the `Viewport`. Each `GameElement` implementation has its own `View` subclass;

Viewport The `Viewport` shows the current game to the player. It is responsible for rendering the maze and delegates rendering of the `GameElements` to the associated `View` instances.

3.3.2 Completing the Pac-Man 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 Pac-Man;
- There is no bonus item that Pac-Man can pick up.

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

3.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pac-Man game is scoring. This concern involves a number of events. The score should be updated whenever Pac-Man eats a bonus, pill, power pill or ghost. And the score itself has to be drawn on the screen to relay it back to the user. These events scatter over multiple classes: `Level` (updating score), `Ghost` (updating score), `Viewport` (drawing score). Thus scoring is an example of a crosscutting concern where only new behavior is added without changing the original behavior of the program.

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 3.2 shows an example Compose⋆ concern definition of scoring.

```
concern Scoring in PacmanTwo
{
  filtermodule registerScorePawns
  {
    externals
      score : PacmanTwo.Scoring.Score =
          PacmanTwo.Scoring.Score.instance();
    inputfilters
      scored : After = { [*.died] score.pawnDied }
  }

  filtermodule registerScoreLevel
  {
    externals
      score : PacmanTwo.Scoring.Score =
          PacmanTwo.Scoring.Score.instance();
    inputfilters
      scored : After = { [*.eatPill] score.eatPill
                       , [*.eatPowerPill] score.eatPowerPill }
  }

  filtermodule renderScore
  {
    internals
      scoreview : PacmanTwo.Scoring.ScoreView;
    inputfilters
      render : After = { [*.renderAll] scoreview.renderScore }
  }

  superimposition
  {
    selectors
      lvl = { C | isClassWithName(C, 'PacmanTwo.Level') };
      pawns = { C | isClassWithNameInList(C, ['PacmanTwo.Ghost']) };
      viewport = { C | isClassWithName(C, 'PacmanTwo.GUI.Viewport' ) };
    filtermodules
      lvl <- registerScoreLevel;
      pawns <- registerScorePawns;
      viewport <- renderScore;
  }
}
```

Listing 3.2: `Scoring` concern in Compose⋆

This concern definition is called `Scoring` (line 1) and contains two parts. The first part is the declaration of the filter modules called `registerScorePawns` (lines 3–9), `registerScoreLevel` (lines 11–18), and `renderScore` (lines 20–26). The first two filter modules `registerScorePawns` and `registerScoreLevel` will update the current score. When Pac-Man eats a ghost the `died` method will be called on the `Ghost`. The filter module `registerScorePawns` contains an *after filter* called `scored` (line 8). This filter will send a message to the `Score` instance after the `died` method has been called. The `registerScoreLevel` filter module contains a similar filter for the `eatPill` and `eatPowerPill` methods. The filter module `renderScore` contains an*after filter* that will send a message to the `renderScore` method of a `ScoreView` instance after the `renderAll` method of `Viewport` has been called. The final part of the concern definition is the superimposition part (lines 28–38). This part defines that the previously defined filter modules `dynamicscoring` are to be superimposed on the selected classes.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by the classes `Score` and `ScoreView`. Listing 3.3 shows an example implementation of class `Score` and Listing 3.4 shows an example implementation of class `ScoreView`. An instance of `Score` will receive messages sent by the `scored` filters in the `registerScorePawns` and `registerScoreLevel` filter modules. The `Score` instance will subsequently perform the events related to the scoring concern. The `Score` instances is declared as *external* and therefor both filter modules will use the same instance of this class. A `ScoreView` instance will render the current score, as stored in the global `Score` instance, on the `Graphics` instance that was passed as an argument of the original method call. The `JoinPointContext` argument that is passed to the `renderScore` method contains information about the join point where the message was intercepted. It provides access to the various properties of a message like the sender and target, but also provides access to the arguments and result of the method that was called. This allows methods called by, for example, the after filter to change the result or use the function arguments for additional processing. For the scoring concern the `JoinPointContext` is only used to get access to the `Graphics` instance to draw the score on.

3.3.2.2 Implementation of Dynamic Strategy

The second system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should hunt or flee, depending on the state of Pac-Man. We can implement this concern by using the strategy design pattern. However, this would require us to modify the existing code. This is not the case when we use Compose* with *dispatch* and *send filters*. Listing 3.5 demonstrates this. Two new `Strategy` implementations have been added to perform the actual decision making. The `Stalker` strategy will find the best direction toward Pac-Man, and the `Flee` strategy will find the best direction away from Pac-Man.

This concern modifies three parts of the `AIController` based on the ghost they are controlling and if Pac-Man is in evil mode. Every time a ghost enters a new cell of the maze is will ponder if it should change direction or simply continue in the same direction, this is done in the `ponder` method of the `AIController`. By default, it will

```
1  package PacmanTwo.Scoring;
2
3  import Composestar.StarLight.ContextInfo.JoinPointContext;
4
5  public class Score {
6    public static final int POINTS_PILL       = 10;
7    public static final int POINTS_POWERPILL  = 50;
8    public static final int POINTS_GHOST      = 200;
9
10   protected static Score _instance;
11   protected int currentScore;
12   protected int ghostEatCount = 0;
13
14   protected Score() {
15     reset();
16   }
17
18   public static Score instance()  {
19     if (_instance == null) _instance = new Score();
20     return _instance;
21   }
22
23   public void reset() {
24     currentScore = 0;
25   }
26
27   public int getScore() {
28     return currentScore;
29   }
30
31   public void setScore(int inval) {
32     currentScore = inval;
33   }
34
35   public void addScore(int inval) {
36     currentScore += inval;
37   }
38
39   public void eatPill(JoinPointContext jpc) {
40     ghostEatCount = 0;
41     addScore(POINTS_PILL);
42   }
43
44   public void eatPowerPill(JoinPointContext jpc)  {
45     ghostEatCount = 0;
46     addScore(POINTS_POWERPILL);
47   }
48
49   public void pawnDied(JoinPointContext jpc)  {
50     ghostEatCount = Math.min(4, ghostEatCount+1);
51     addScore(POINTS_GHOST * ghostEatCount);
52   }
53 }
```

Listing 3.3: Implementation of class `Score`

```
1  package PacmanTwo.Scoring;
2
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import Composestar.StarLight.ContextInfo.JoinPointContext;
6
7  public class ScoreView {
8    protected Score score;
9
10   public ScoreView() {
11     score = Score.instance();
12   }
13
14   public void renderScore(JoinPointContext jpc) {
15     Graphics g = (Graphics)jpc.GetArgumentValue((short)0);
16     g.setColor(Color.YELLOW);
17     g.drawString("Score:", 492, 40);
18     g.drawString(""+score.getScore(), 492, 50);
19   }
20 }
```

Listing 3.4: Implementation of class `ScoreView`

only make a new decision at a given interval or when it bumped into a wall. Not all ghosts are equal: one ghost is very smart, one is just smart, and the other two are not smart at all. The `smarty` filter (line 15) will make sure the very smart ghost will always make a new decision when it enters a new cell. The `setstrat` filter (line 17) will modify the movement strategy. If Pac-Man is not evil and if the ghost is smart or very smart (line 18) it will redirect the movement decision to the `Stalker` strategy. If Pac-Man is evil (line 19) it will redirect the decision to the `Flee` strategy.

In the end the very smart ghost will be on Pac-Man's heels, the smart ghost will try to move in Pac-Man's direction but not as well as the very smart ghost does. The other two ghosts will move in a random direction. When Pac-Man is evil all ghosts will flee from Pac-Man, the very smart ghost will perform best in fleeing from Pac-Man because it will ponder for a new move in every cell of the maze.

3.3.2.3 Implementation of the Bonus Pickup

The last system requirement is an example of a large cross cutting concern that needs to add behavior to various major components of the game. The behavior of the bonus

```
concern DynamicStrategy in PacmanTwo
{
  filtermodule dynstrat
  {
    internals
      stalker : PacmanTwo.Strategy.Stalker;
      chicken : PacmanTwo.Strategy.Flee;
    externals
      game : PacmanTwo.Game = PacmanTwo.Game.instance();
    conditions
      isEvil : game.hasEvilPacman();
      isSmart : inner.isSmart();
      isVerySmart : inner.isVerySmart();
    inputfilters
      smarty : Dispatch = { isVerySmart => [*.ponder] inner.getNextMove }
    outputfilters
      setstrat : Send = {
        !isEvil & isSmart => [*.doGetNextMove] stalker.getNextMoveNS ,
        isEvil => [*.doGetNextMove] chicken.getNextMoveNS
      }
  }

  superimposition
  {
    selectors
      ai = { C | isClassWithName(C, 'PacmanTwo.AIController') };
    filtermodules
      ai <- dynstrat;
  }
}
```

Listing 3.5: `DynamicStrategy` concern in Compose⋆

pickup is as follows:

1. The bonus pick up will be placed in the maze after a set delay;
2. A second bonus pick up is placed after a set delay after the first bonus was picked up;
3. There are only two pick ups per maze.
4. Only Pac-Man can pick up the bonus, ghosts will ignore it;
5. The number of points scored for the bonus will increase every new maze;
6. The bonus is rendered differently depending on the points that can be scored by picking it up.

A few new classes will need to be introduced. First, a new `GameElement` called `BonusPickup` is needed to represent the bonus pick in the maze. And a `View` subclass called `BonusView` is needed to render the `BonusPickup` on the screen. In order to implement the new behavior certain messages need to be intercepted, as shown in Listing 3.6. The overall management of the bonus behavior is handled by the singleton class `Bonus`.

The first (lines 3–10) event the bonus has to react on is the `startNewGame` in the `Game` class. The `Bonus` class will use this even to initialize the bonus management and start the timer for placing the first `BonusPickup`. The `tick` event is also intercepted to update the timer in the `Bonus` class. When the timer reaches zero the bonus manager will the `BonusPickup` to the current maze. Next the game has to react on Pac-Man touching the `BonusPickup` (lines 12–18). Because `BonusPickup` is a stationary element it will not check if it has been touched by an other element, therefor the touching check has to be performed elsewhere. The `touch` on Pac-Man will be forwarded to the `pacmanTouch` method of the `Bonus`. This method should check if the touched `GameElement` is an instance of `BonusPickup` because the same event will be triggered when Pac-Man touches any other `GameElement` instance. `pacmanTouch` will, if the `BonusPickup` was touched, update the score, and reset the timer for the second bonus. For the registration of the score the `Score` class that was introduced in 3.3.2.1. The `Score` singleton class will simply be reused for the scoring. The third filter module (lines 20–26) will act on the `createViews` method of `Viewport`. This event will be called when the views for all game elements are registered, standard only `View` classes for Pac-Man and the ghosts are registered. `Bonus` will use this to register the `BonusView` with the `Viewport` instance so that the `BonusPickup` will also be rendered on the maze. An other requirement for the bonus pickup is that the bonus reacts to maze changes in order to change the shape of the `BonusPickup` and the number of points gained from picking it up. To keep track of the current maze number a level counter is used in the `Bonus` class that is reset by the `startGame` method. The `LevelUp` filter module (lines 28–34) intercepts the `getNewMaze` of the `Level` class, this method is called when a new level is created. When the `getNewMaze` method was called the bonus manager will increase the level counter. When the next bonus pickup is created it will reflect the current level number.

With these changes in place the Pac-Man game meets the initially set requirements without modifying the source code of the basic implementation.

```
concern BonusConcern in PacmanTwo
{
  filtermodule BonusManager
  {
    externals
      bm: PacmanTwo.Bonus.Bonus = PacmanTwo.Bonus.Bonus.instance();
    inputfilters
      startBonus : After = { [*.startNewGame] bm.startGame };
      tick : After = { [*.tick] bm.tick }
  }

  filtermodule TouchBonus
  {
    externals
      bm: PacmanTwo.Bonus.Bonus = PacmanTwo.Bonus.Bonus.instance();
    inputfilters
      touched : After = { [*.touch] bm.pacmanTouch }
  }

  filtermodule RegisterBonusView
  {
    externals
      bm: PacmanTwo.Bonus.Bonus = PacmanTwo.Bonus.Bonus.instance();
    inputfilters
      touched : After = { [*.createViews] bm.createViews }
  }

  filtermodule LevelUp
  {
    externals
      bm: PacmanTwo.Bonus.Bonus = PacmanTwo.Bonus.Bonus.instance();
    inputfilters
      lvlup : After = { [*.getNewMaze] bm.levelUp }
  }

  superimposition
  {
    selectors
      game = { C | isClassWithName(C, 'PacmanTwo.Game') };
      pacman = { C | isClassWithName(C, 'PacmanTwo.Pacman') };
      viewport = { C | isClassWithName(C, 'PacmanTwo.GUI.Viewport') };
      level = { C | isClassWithName(C, 'PacmanTwo.Level') };
    filtermodules
      game <- BonusManager;
      pacman <- TouchBonus;
      viewport <- RegisterBonusView;
      level <- LevelUp;
  }
}
```

Listing 3.6: `BonusConcern` concern in Compose⋆

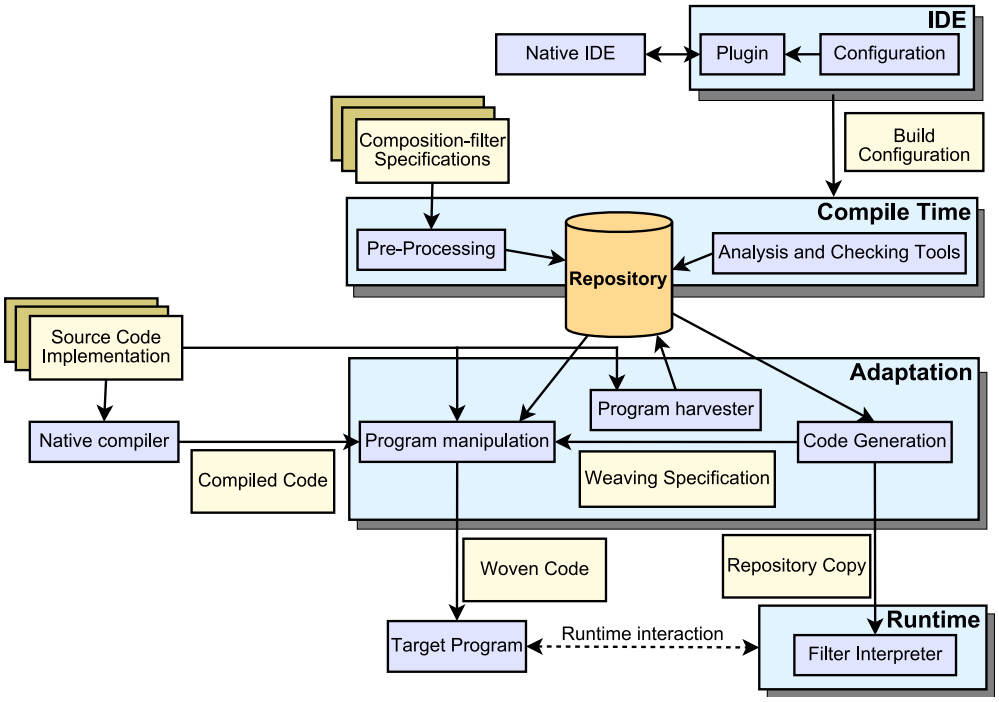


Figure 3.3: Overview of the Compose⋆ architecture

3.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 3.3. The Compose⋆ architecture can be divided in four layers [42]: IDE, compile time, adaptation, and runtime.

3.4.1 Integrated Development Environment

Some of the purposes of the Integrated Development Environment (IDE) layer are to interface with the native IDE and to create a build configuration. In the build configuration it is specified which source files and settings are required to build a Compose⋆ application. After creating the build configuration the compile time is started.

The creation of a build configuration can be done manually or by using a plug-in. Examples of these plug-ins are the Visual Studio add-in for Compose⋆/.NET and the Eclipse plug-in for Compose⋆/J and Compose⋆/C.

3.4.2 Compile Time

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

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, which is called the 'repository'. This knowledge base contains the structure and metadata of the program. Different modules can use this knowledge base to execute their activities. Examples of modules within analysis and validation are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and validation of the super imposition and its selectors.

3.4.3 Adaptation

The adaptation layer consists of the program manipulation, harvester, and code generator. These components 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 adding this information to the knowledge base. The code generation generates a reduced copy of the knowledge base and the weaving specification. This weaving specification is then used by the weaver, which is contained in the program manipulation component, to weave in the calls to the runtime into the target program. The end result of the adaptation layer is the woven target program. This program interfaces with the runtime.

3.4.4 Runtime

The runtime layer is responsible for executing the concern code at the join points. It is activated at the join points by function calls that are woven in by the weaver. A reduced copy of the knowledge base, containing the necessary information for filter evaluation and execution, is enclosed with the runtime. Method calls are intercepted by the runtime. The runtime evaluates the filter set for the intercepted message and executes the corresponding filter actions. The runtime also facilitates the debugging of the composition filter implementations.

3.5 Platforms

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

Compose⋆/.NET targets the .NET platform. It is the oldest implementation of Compose⋆. Its weaver operates on CIL byte code. Compose⋆/.NET is programming language independent as long as the programming language can be compiled to CIL code. An add-in for Visual Studio is provided for ease of development. Compose⋆/J targets the Java platform and provides a plug-in for integration with Eclipse. Compose⋆/C contains support for the C programming language. The implementation is different

from the Java and .NET counterparts, because it does not have a run-time environment. The filter logic is woven directly in the source code. Because the C language is not based on objects, filters are woven on functions based on membership of sets of functions. Like the Java platform, Compose⋆/C provides a plug-in for Eclipse.

3.6 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 a concern. Compose⋆ offers three features that use this possibility, which originate in more control and correctness over an 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 runtime. 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 lets method `m` continue and the second filter only accepts for methods `a` and `b`. Because only method `m` can reach the second filter, it never accepts. This might indicate a conflict.

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 not the case for the meta filter, its user-defined, and therefore unpredictable, behavior poses a problem for the analysis tools.

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

Integrated Development Environment support
: The Compose⋆ implementations all have an IDE plug-in; Compose⋆/.NET for Visual Studio, Compose⋆/J and Compose⋆/C 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.

Some language properties of Compose⋆ can also be seen as features, being:

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.

3.7 StarLight

In 2006, development began on a lightweight branch of Compose⋆/.NET, called StarLight. The aim of this new branch is to provide a more robust and efficient variant of Compose⋆. Certain more advanced features, such as multiple inheritance, have been excluded from this new branch, while new features have been implemented to meet industry demands. This section describes the differences in the architecture between Compose⋆ and StarLight and certain new features of StarLight.

3.7.1 StarLight Architecture

This section describes differences between the StarLight Architecture and the Compose⋆ architecture, as shown in Figure 3.3.

Integrated Development Environment
StarLight has its own Visual Studio integration. This integration provides the following functionality:

- A project service to open and use StarLight projects in Visual Studio;
- Syntax highlighting and IntelliSense for concern files;
- MSBuild is used to analyze and weave the concerns.

Compile Time
StarLight uses the same compile time as Compose⋆. An inlining engine is added to the compile time. This inlining engine translates the filter set to a platform independent abstract instruction model for each individual message. This abstract

instruction model represents a procedural structure. It can be easily translated to program code for a specific procedural platform.

Adaptation
StarLight has its own adaptation layer. It contains an analyzer that creates the language model. It also contains a weaver that translates the abstract instruction models for each message to IL code and weaves this code in the appropriate places of the target assemblies.

Runtime
The StarLight version does not have a runtime, because a complete filter set translation is woven in the target program.

3.7.2 Explicit Modeling of the Returning Flow

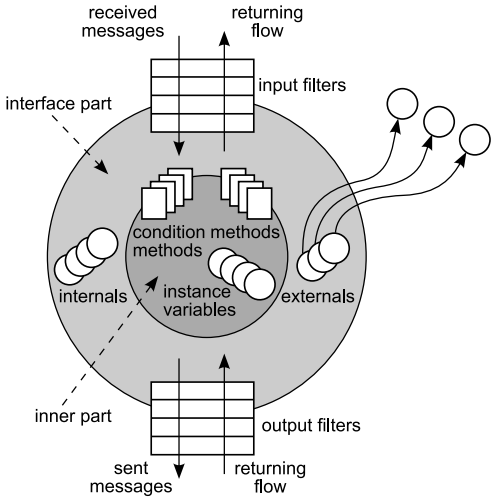


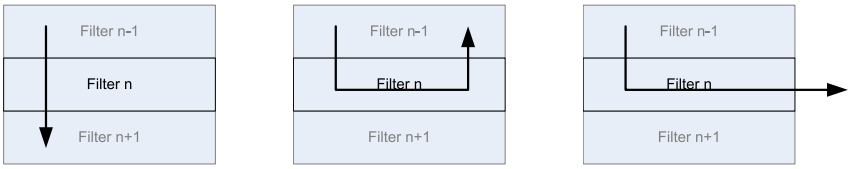
Figure 3.4: Explicit modeling of the returning flow

Originally, composition filters only modeled the calling flow: sending the message from the sender to the target. However, there is also a returning flow: the action of returning the control to the sender after the target has ended the execution of the message. This returning flow possibly contains a return value, but this is not necessary. In the new StarLight implementation we also want to model this returning flow explicitly. This makes it possible to execute filter actions after the message has been dispatched.

Figure 3.4 shows the returning flow in the composition filters model. Both input filters and output filters have a returning flow.

When is the Flow Returned? Explicit modeling of the returning flow raises questions such as when is the flow returned and is there always a returning flow. From the

composition filters perspective, it is the filter action that decides whether to return the flow or to continue to the next filter. There are actually three possibilities, as shown in Figure 3.5.



(a) Continue (b) Return (c) Exit

Figure 3.5: Filter actions can either continue, return or exit.

Continue
The flow continues to the next filter. Examples of filter actions that continue the flow are the `Substitution` action and the `Advice` action.

Return
A filter action can return the flow. In this case, the calling flow turns into a returning flow. An example of a filter action that returns the flow is the `Dispatch` action.

Exit
The third option available for filter actions is to exit the filter set. This equals an exception or abnormal return. An example of a filter action that exits the filter set is the `Error` action.

4-Action Filter Types Because StarLight models the returning flow explicitly, it is possible to execute actions on the returning flow. The actions that are executed, are specified by the filter type. Therefore, StarLight has 4-action filter types instead of 2-action filter type.

	Calling Flow	Returning Flow
Accept	accept-call action	accept-return action
Reject	reject-call action	reject-return action

Table 3.1: The four filter actions of a filter type.

Evaluation of Filters on the Returning Flow? The filters are not evaluated on the returning flow. They are only evaluated on the calling flow. The filter action that is executed by a filter on the returning flow depends on whether the filter accepted or rejected on the calling flow.

The actions on the returning flow are executed in reverse order of the filter set.

3.7.3 Defining New Filter Types and Filter Actions

StarLight provides a new way to define new filter types and new filter actions. Actually, there are no primitive filter types and filter actions anymore, as there are in Compose*. However, there are certain common filter types and filter actions provided by the StarLight API. But they are defined in the same way as any other new filter type or filter action.

```
1 [FilterActionAttribute("LoggingInAction",
      FilterActionAttribute.FilterFlowBehavior.Continue,
      FilterActionAttribute.MessageSubstitutionBehavior.Original)]
2 public class LoggingInAction : FilterAction
3 {
4   public override void Execute(JoinPointContext context)
5   {
6     //Logging-in implementation
7   }
8 }
```

Listing 3.7: Defining the `LoggingIn` filter actions

New filter actions can be defined by extending the `FilterAction` class. Listing 3.7 shows an example of a new filter action. By overriding the `Execute` method, the action can be implemented. When the action is executed at runtime, a call is done to its `Execute` method. A mandatory custom attribute on the filter action specifies the following information:

- The name of the filter action;
- The flow behavior of the filter action: continue, return or exit;
- The substitution behavior of the filter action. This specifies whether the message continues substituted or not after the filter action. An example of a filter action that leaves the message substituted is the `Substitution` action.

```
1 [FilterTypeAttribute("Logging", "LoggingInAction",
      FilterAction.ContinueAction,
      "LoggingOutAction",FilterAction.ContinueAction)]
2 class LoggingFilterType : FilterType
3 {
4   //No implementation
5 }
```

Listing 3.8: Defining the `Logging` filter type

A new filter type can be defined by extending the `FilterType` class. Listing 3.8 shows an example of a new filter type. This class has no implementation methods, as it is not used at runtime. It is only used to specify a new filter type at compile time. A mandatory custom attribute specifies the following information:

- The name of the filter type. This name can be used in filter specifications;
- The filter actions, in the order accept-call, reject-call, accept-return and reject-return.

The example shows the definition of a `Logging` filter type that logs both on call as on return, when the filter accepts.

3.7.3.1 Built-in Filter Actions

The StarLight API provides several filter actions:

Continue This filter action continues the execution of the filter set to the next filter. It does not have any other behavior

Dispatch This action dispatches the message to the specified target.

Advice This action executes a specific advice method, specified by the substitution part. This is a lightweight replacement of the `Meta` action. It cannot change the message or change the flow behavior of the message. It also does not have the multithreading functionality of the `Meta` action. After an `Advice` action, the flow continues to the next filter with an unchanged message.

Error This action raises an exception and causes the flow to exit the filter set.

Substitution This action substitutes the message with the message specified by the substitution part. The execution of the filter set continues to the next filter.

3.7.3.2 Built-in Filter Types

The StarLight API provides several filter types:

		Calling Flow	Returning Flow
Dispatch	Accept	Dispatch action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Before	Accept	Advice action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
After	Accept	Continue action	Advice action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Error	Accept	Error action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Substitution	Accept	Subst. action	Continue action
	Reject	Continue action	Continue action

3.7.4 Conditional Superimposition

Another new feature in the StarLight version of Compose⋆ is conditional superimposition. Conditional superimposition makes it possible to superimpose a filter module conditionally. At runtime the condition is evaluated to decide whether the filter module should be executed. This makes it possible to turn crosscutting behavior on and off at runtime. It is even possible to turn the filter module off for specific classes, because the condition can get the classname as parameter.

```
1  concern LoggingConcern
2  {
3    filtermodule LoggingFM
4    {
5      inputfilters
6        logging  : Logging = { True => [*.*] }
7    }
8
9    superimposition
10   {
11     conditions
12         loggingEnabled : Logging.LoggingEnabled;
13     selectors
14       loggingClasses = { ... };
15     filtermodules
16       loggingEnabled => loggingClasses <- LoggingFM;
17   }
18 }
```

Listing 3.9: Conditional superimposition example

Listing 3.9 shows an example of conditional superimposition. In this example, the `LoggingFM` filter module is superimposed conditionally. Before the filter module is executed at runtime, the condition `loggingEnabled` is checked.

Chapter 4

Signature modification

A technique that is introduced by composition filters is signature modification. In short, signature modification allows composition filters to add and remove methods to the signature of types. These modifications can be made at compile-time and runtime.

We begin with an explanation of what is signature modification. We then look at the problems that are the result of introducing signature modification to Compose⋆/.NET. A solution was introduced to provide support for signature modification in Compose⋆/.NET, which is described in Section 4.3. Unfortunately, the old solution also introduced a number of problems, which are described in the last section.

4.1 Introduction

All signature modification in Compose⋆/.NET is applied to .NET types. Types in the .NET framework can be described as units that contain data and/or operation members. Types in the framework include simple value types, such as boolean and integer, but signature modification is applied to user-defined types. User-defined types include classes, interfaces and structs.

The signature of a type is the collection of a type's visible members. The signature defines the possible interactions with an object of the type. A good object-oriented programming practice is to only allow operations (methods) to be visible members. Therefore, signature modification can only be applied to operations that are part of the signature of a type.

Signature modification can be described as modifying the signature of a type by adding a method to or removing a method from a type. Adding a method to a type is called signature expansion. Removing a method from a type is called signature reduction.

Figure 4.1 shows an example of both signature expansion and signature reduction applied to a class. The first concern applies signature reduction by removing the `TestReduction` method from the `Example` class. The second concern applied signature expansion by adding the `TestAddition` method to the `Example` class. A code example is described in Listing 4.1 and Listing 4.2.

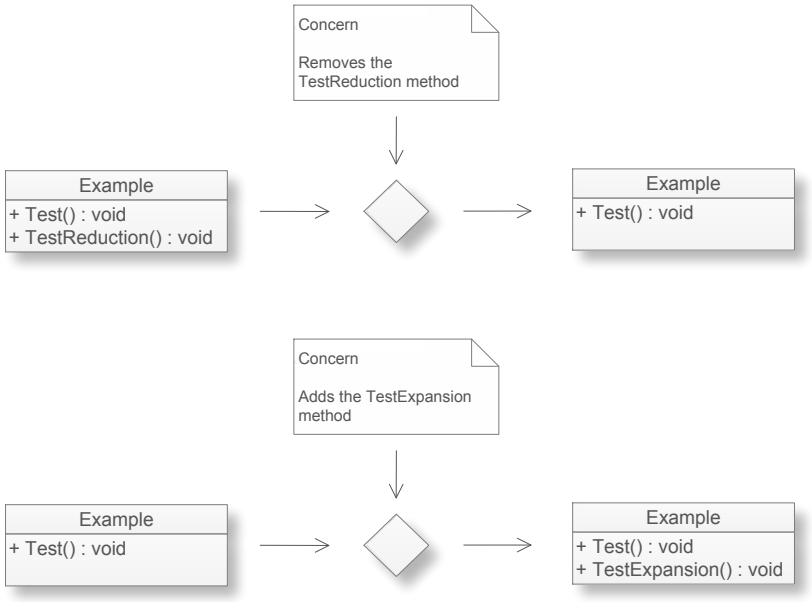


Figure 4.1: Example of signature modification

Listing 4.1 shows the code of the concern `SignatureModification` that applies both signature reduction and signature expansion. Signature reduction is applied by the `reduction_filter`, which applies an Error filter on a method in the targeted type (`SignModTestClass`). If the Error filter is the only or the first filter applied on a method, then the method will be removed from the signature.

Signature expansion is applied by the `expansion_filter`, which applies an Dispatch filter. Because the selector target (`*.TestExpansion`) is not part of the signature of `SignModTestClass`, it is added by the composition filter to the signature of the type. The substitute target (`test_expansion.TestExpansion`) references to the method that implements the functionality of the added method.

The example in Figure 4.1 only shows static signature modification which are applied at compile-time. A variant of signature modification is conditional signature modification. A condition is added to the filters in the concern, which is evaluated at runtime. The outcome of this evaluation determines whether or not a call to the target method is allowed.

4.2 Problems introduced by signature modification

There is one major problem that prevents us from simply introducing signature modification to Compose⋆/.NET. Signature modification is not supported by the .NET compilers that are used to compile the source code. An example of source code that cannot be compiled by the compilers is shown in Listing 4.1 and Listing 4.2.

```
concern SignatureModification in SignModTestClass {
  filtermodule signatureModification {
    internals
      test_expansion : Example.ExpansionTest;
    inputfilters
      reduction_filter : Error = {[*.TestReduction]};
      expansion_filter : Dispatch = {[*.TestExpansion]
          test_expansion.TestExpansion}
  }
  superimposition {
    selectors
      testing = { C | isClassWithName(C, 'Example.SignModTestClass') };
    filtermodules
      testing <- signatureModification;
  }

  implementation in CSharp by SignatureModification as "ExpTest.cs" {
    namespace Example {
      public class ExpansionTest {
        public void TestExpansion() {
          /* */
        }
      }
    }
  }
```

Listing 4.1: Signature modification concern in Compose⋆

```
namespace Example {
  class SignModTestClass {
    public void TestReduction() {
      /* */
    }
  }

  class Test {
    public static void Main() {
      SignModTest test = new SignModTest();
      test.TestExpansion();
      test.TestReduction();
    }
  }
}
```

Listing 4.2: Example of signature modification in C#

Listing 4.1 shows a concern file that applies both signature expansion and signature reduction. Both signature modifications are applied on the `SignModTestClass` class that is described in Listing 4.2. The signature reduction removes the `TestReduction` method from the class. The signature expansion adds the `TestExpansion` method to the class.

The problems at compile-time revolve around the calls `test.TestExpansion()` and `test.TestReduction()`. Because the compiler does not take the composition filter into account, it will base the type definition on the source file alone. This leads to two kinds

of undesired behavior. First, the call to `TestExpansion` fails since this method cannot be found in the type definition. Second, the call to `TestReduction` is found to be valid since it is not removed by the composition filter and is therefore still present in the type definition.

4.3 Old solution

A solution was found and implemented that added support for signature modification to Compose⋆/.NET while still using the standard .NET compilers. The found solution involves creating dummy version of the orginal source files. These dummy versions contain the types with their signatures from the original source files, but none of the functionality defined in the orginal files. Signature modification can now be applied on the signatures in the dummy sources, before the dummy sources are compiled to an assembly. This dummy assembly is then referenced during the compilation of the original source files.

A more extensive description of the solution can be found in [20].

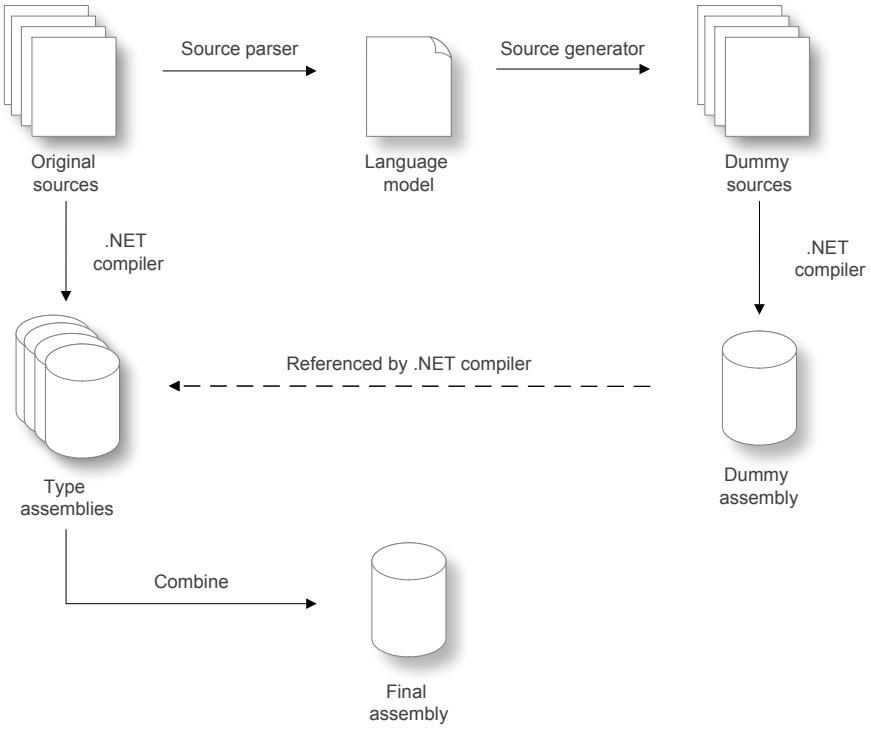


Figure 4.2: Overview of the old solution

Figure 4.2 shows an overview of the old solution that was implemented. We will describe the major components of the solution in more detail:

Source parser The first step in the compilation process consists of a source parser that takes the original source files and extracts the necessary type information. The extracted type information is represented in a language model, which allows us to represent the source code in a language independent manner.

Source generator The second step is the source generator, which generates a dummy version of each of the original source code files. The required source code information is taken from the language model, created in the previous step. The dummy version of a source code file contains the signature of the types in the source code file without the implementation, but with signature modification applied.

.NET compiler The .NET compilers are used in two steps. First, the compiler is used to compile the dummy source files into a dummy assembly. This dummy assembly contains the same information as the dummy source files, but in IL. Second, the compiler is used to compile the original source code files. The original source code files are compiled one at a time.

The compiler references the dummy assembly during the compilation of a source code file, to provide the type information found in the other original source code files. The dummy assembly also contains the types in the original source code file, but the compiler will choose the type information in the source file over the information in the dummy assembly.

Combine assemblies Because the original source files need to be compiled one at a time, the compilation step results in a number of assemblies. Each of these assemblies contain the fully implemented types of one of the original source files and the dummy types of the other source files. These assemblies need to be combined into one fully implemented assembly.

4.4 Problems introduced by the found solution

Although the found solution allowed support for signature modification in Compose⋆/.NET, it did introduce a number of problems. The three biggest problems are described here in more detail.

4.4.1 Incompatibility with .NET 2.0 compilers

The biggest problem was encountered when a new version of Compose⋆/.NET was introduced that would support .NET 2.0. The solution that provided support for signature modification was found to be incompatible with the new .NET 2.0 compilers.

Listing 4.3 shows an example of source code that cannot be compiled by the old solution when using the .NET 2.0 compilers. The problem manifests itself when we are compiling the source file *B.cs*. Class `B` contains the method `TestA`, which defines

```
1 namespace Example {
2   public class A {
3     public void Test(B b)
4     {
5       /* */
6     }
7   }
8 }
```

(a) A.cs

```
1 namespace Example {
2   public class B {
3     public void TestA()
4     {
5       A a;
6       a.Test(this);
7     }
8   }
9 }
```

(b) B.cs

Listing 4.3: Example of source code that cannot be compiled with the old solution

an object of class `A` and calls the method `Test` with a reference to itself (`this`) as a parameter.

Because we are only compiling the *B.cs* source file, the type information for class `A` is taken from the referenced dummy assembly. Class `A` in the assembly does contain the called `Test` with an object of class `B` as a parameter. However, the class `B` expected as a parameter is the class `B` defined in the dummy assembly and not the class `B` that is passed as a parameter, which is defined in *B.cs*.

If a .NET 1.1 compiler would encounter different definitions of a single type, it would issue a warning and assume one of the type definitions to be the correct one. If a .NET 2.0 compiler encounters this situation, it halts compilation and issues an error. This prevents the original source files from being compiled and stops the Compose⋆/.NET compilation process.

4.4.2 Language dependencies

There are two components in the old solution that are language dependent: the source parser and the source generator. Both components are required to have an implementation for each programming language that is supported by Compose⋆/.NET, since both source parsers and source generators generally provide support for only one programming language.

This has an adverse effect on the maintainability of the Compose⋆/.NET project. Any new functionality introduced by a new version of the .NET framework might have to be implemented at multiple locations in the project.

If a new programming language were to be supported by Compose⋆/.NET, it would require a new implementation of the source parser and possibly the source generator. This in turn would further increase the amount of source code and therefore reduce the maintainability of the project.

4.4.3 Self-referencing types

The last major problem in the old solution revolves around self-referencing types. Self-referencing types are types which call one or more methods that are defined in their own type. An example of a self-referencing type is described in Listing 4.4.

```
1  namespace Example {
2    class SelfReferencingClass {
3      public void Test2() {
4        /* */
5      }
6
7      public void Test() {
8        Test2();
9      }
10   }
11 }
```

Listing 4.4: Example of a self-referencing class

The problem occurs when the referenced method is part of the signature of the type. If the method is part of the signature, it is possible that the method is removed by a composition filter. This would mean that the call to the method should fail.

However, this signature modification is applied to the dummy version of the self-referencing type, not the original source file. The compiler uses the type definition from the source file, rather than the type definition in the dummy assembly. This means that a self-referencing call to a method that has been removed by a composition filter will be allowed by the compiler, since the method is still present in the original and therefore in the used type definition.

The same problem occurs for self-referencing call to methods that have been added by a composition filter. This method will only be added to the type definition in the dummy assembly, but not to the type definition used by the compiler (the type definition as defined in the source file). Therefore, any self-referecing calls to added methods will fail to compile.

Chapter 5

Research into new solutions

This chapter describes the research into new solutions in an attempt to resolve the problems encountered in the old solution. As was stated in the previous chapter, the standard .NET compilers do not take signature modification into account and a work-around for this problem has to be part of the new solution.

To find a new solution, we look at all parts of the standard .NET compilation process and the old solution. We have identified three parts in both compilation processes:

- The orginal source files.
- The used compiler.
- Dummy generation.

To find a new solution, we look at each of the three identified parts and present modifications to them that will allow signature modification to be applied by composition filters. A description of each part and the suggested modifications can be found in the next three sections. This is followed by an overview and evaluation of the found solutions. From this evaluation, we can deduce a new solution which is described in Section 5.5. Finally, we evaluate the new solution; it is compared against the problems identified in the old solution and any problems introduced by the new solution are identified.

The text in this chapter is largely based on the text found in "New signature expansion process for Compose⋆ " [24].

5.1 Modifying the source files

Modifying the source code allows any signature modification to be applied by Compose⋆.NET directly to the source code before compilation. This results in source code that takes the composition filters into account and that can be compiled by the standard compilers.

Source code modifications have two major advantages. First, since all the required modifications are made to the source code before the compilation process, the source

modification will work with the standard .NET compilers. Second, because the signatures of the types are modified before compilation, self-referencing types are supported. Self-referencing types get the type information from their own source file. Because this type information was modified before compilation, compilers have the proper type information and calls will be handled properly.

There are two ways to modify the source code to provide support for signature modification. First, find a language construction that is already in the supported languages and the CLR to include support for signature modification. Second, modify the original source code at compile-time with an external tool to include signature modifications before the compiler compiles the code.

Existing language constructions

An example of a language construction already present is the *Reflection* namespace, which is part of the .NET framework. By calling a method through a Reflection method, we circumvent the type-checking by the compiler and the code will compile. The composition filters can then be applied at runtime to determine whether the method call is actually allowed or not.

The big advantage of the first method is that it does not require any extra actions during the compilation process. Since we are using a construction that is already present in the used language and in the CLR, it is already supported by the compiler.

A big disadvantage of this method is that the developer has to use a special language construction in the source code. This means that composition filters cannot be used transparently on just any source code. A second disadvantage is that the language construction is not guaranteed to be supported by all .NET programming languages. A third disadvantage is that a special language construction will circumvent type-checking by the compiler, which can lead to more errors at runtime. A final disadvantage is a decreased performance at runtime.

Source code modification

The second method requires a tool that parses the source code and modifies it, thus applying the composition filters. The big advantage of this method is that it is completely transparent to the developer. The modifications to the source code are made at compile-time before the standard .NET compiler actually compiles the source code.

A second advantage is that the type-checking functionality of the compilers can be used. The compiler can check the modified source code and warn if any of the modified source code cannot be compiled. This includes the checking of types that contain self-referencing calls, since the compiler will find calls to methods that no longer exist in the source code.

The biggest disadvantage of this method is that it requires a code parser to be present for every language that Compose⋆.NET supports, making this solution language-dependent. A second disadvantage follows from the language dependency: writing and maintaining a code modification tool for every supported language is a large task.

An alternative is to use existing code parsers. There are two problems with using existing parsers. The first problem is that modifying the source code cannot be done through most code parsers. This means that a tool to modify the code is still necessary (and one per language again). The second problem is the availability of high quality code parsers. Although there are a lot of code parsers to be found, very few of them are of a quality that would make them viable solutions. Examples of code parsers and code parser generators can be found in [36, 44, 55].

5.2 Changing compilers

The first approach to changing compilers would be to modify the existing .NET compilers. Unfortunately, this is impossible because Microsoft has not provided the source code of its .NET compilers and does not allow the modification of its compilers in any way. Therefore, if we want to change the compilers, we must replace the standard .NET compilers with custom compilers that support composition filters.

Replacing the standard .NET compilers with custom compilers has one advantage: there is no need to modify any other part of the compilation process. If the compiler takes any signature modification into account, then the compiler can use the original source code. Furthermore, self-referencing types will be supported and other parts of Compose★.NET could be integrated into the compilers.

There are two approaches to changing the compilers used in Compose★.NET. First, we can replace the standard .NET compilers with new compilers, written from scratch. The second approach is to modify existing alternative compilers with support for composition filters.

Create new compilers from scratch

Writing the compilers from scratch would result in compilers that are tailor fit for our demands. Any support for composition filters could be integrated into the design of the compilers from the start.

However, writing and maintaining compilers is very time-consuming. Multiple compilers will have to be written; one for each programming language that is to be supported. The resulting code has to be able to run on a standard .NET Framework and updates to the .NET Framework will have be incorporated into the compilers. Writing the compilers from scratch is therefore an impractical option.

Modify existing compilers

There is an open source compiler that could be modified for Compose★.NET. This is the compiler that is part of the Mono project. Besides being open source, the Mono project has another advantage; it is multi-platform. Since Mono is open source, it is possible and legal to modify the compiler and any other part of the project. This means that it is possible to extend the compiler to include support for composition filters.

There are a number of disadvantages to using the Mono project. First, modifying a compiler is simpler than writing a compiler from scratch, but it is still a time-consuming task. The same goes for maintaining the modified code; every update to the standard Mono project could have consequences for the modified project. A third problem is that the .NET framework versions supported by the Mono project are not the most recent. Finally, the Mono project currently has a compiler for only one programming language: C#. This would restrict all projects using Compose⋆.NET to this one language. The last two problems could be resolved with Mono 2.0, which is due to be released.

5.3 Dummy files generation

The generation of dummy files is already in use in the old solution. There are two types of dummy files that can be generated: dummy source files and dummy assemblies. The two types of dummies contain the same information, but in different formats. The dummy source files contain source code in a programming language whereas the dummy assemblies contain the information in CIL.

As was explained in Section 4.4.1; the dummy assemblies are responsible for the incompatibility of the old solution with the .NET 2.0 compilers. Because there is no easy solution for this problem and because dummy source files can be a viable alternative, we discard using dummy assemblies in favor of a dummy generation solution that uses dummy source files.

The dummy generation process is the same for both dummy types. We analyze the original source file and extract the required type information. The type information is modified to include any signature modifications. The type information is then used to create the dummies. The dummies are used during the compilation of the original source files.

The advantage of dummies generation is that no modifications to the original sources or the .NET compilers are required. A solution with dummy files generation can use the standard .NET compilers, making the solution more transparent for the developer. Dummy files generation has the disadvantage of introducing new steps in the compilation process and, possibly, requiring a number of external tools.

A second disadvantage of using dummy files is the lack of support for self-referencing types. Self-referencing types take the type information from their own source file. Because this file remains unmodified, it does not contain the information with the signature modifications applied. Therefore, the compiler will not take signature modification for self-referecing types into accoutnt at compile-time

The dummy files generation process contains two major steps:

- The extraction of the type information from the original source files.
- The generation of the dummy files from the type information.

5.3.1 Type information extraction

The first step, type information extraction, analyzes the orginal source files and extracts the necessary information from them. This information is used in the next step to generate the dummy files. There are two approaches to extracting the necessary information from the original source files.

Using an external tool

The first approach, retrieving the information from the source code, requires a parser to extract the information. Using a parser gives the same disadvantages as were described when discussing source modification with an external tool (see Section 5.1). This means that a parser has to be built or an existing parser has to be used and a different parser is needed for every supported programming language.

Using Visual Studio

The second approach uses Visual Studio to retrieve the necessary information. Microsoft offers an interface to the Visual Studio environment and through this interface we can retrieve a representation of a source code file. The big advantage of this representation of the source code is that it is independent of any programming language. It can represent any of the languages supported by Microsoft (C#, J# and Visual Basic.NET, Visual C++.NET is also supported though not in the same manner as the other languages).

The main disadvantage of this approach is that it uses Visual Studio to retrieve the information. This would make Visual Studio a requirement when using Compose⋆.NET. The dependency on Visual Studio has another disadvantage: there is no guarantee that the interface will continue to be supported by Microsoft in its current form. The interface could be removed in the future or could return in a completely different form, although the chances of this happening in the near future are remote. The interface is still offered for the latest version of Visual Studio; Visual Studio 2008.

5.3.2 Generation of the dummy files

The type information extracted in the previous step can be used to generate the dummy source files. This step consists of converting the type information to source code in the used programming language and saving the source code to the dummy files. There are two approaches to creating the required dummy files.

Create dummy files using an external tool

The first approach is to use an external tool to create the dummy files. This tool takes the found type information and converts this to source code. It saves the source code to the dummy files.

Using an external tool shares the disadvantages of source code modification with an external tool: a code generation tool is required per support programming language and writing and maintining the tool is a time-consuming task.

Create dummy files using the .NET framework

The second approach to creating dummy files is to use the .NET framework itself. The .NET framework provides support for writing source code via *CodeDOM*. This namespace contains classes, etc. that can be used to represent the logical structure of source code independent of any programming language [38]. The CodeDOM namespace can be used to create a code tree that represents the source code that we want to build.

Using the CodeDOM namespace to create the dummy files has two advantages. First, representing the source code using a CodeDOM tree allows this approach to be independent of any programming language. Functionality to convert the CodeDOM tree to actual source code is provided by Microsoft for a number of .NET programming languages (C#, J# and Visual Basic.NET).

A disadvantage of using CodeDOM is the reliance on Microsoft to keep the CodeDOM namespace, relatively, unmodified for future and present version of the .NET framework. Should Microsoft modify the CodeDOM approach significantly or should CodeDOM be removed or replaced, then this approach is no longer a valid solution. The chance of this is happening is small however: the most recent version of the .NET Framework (version 3.5) still contains the CodeDOM namespace.

5.4 Overview found solutions

To determine if there is a solution that enables signature modification support in Compose⋆.NET and which solution that is, we need to compare the found solutions on a number of criteria. The following criteria are used for the comparison of the different solutions:

Transparency (TR). Whether the solution requires modifications of the original source code.

Language dependency (LD). Whether the solution needs to be implemented per programming language.

Maintainability (MA). The amount of work required to maintain the implementation of the solution.

Forwards compatiblity (FC). The amount of work required to adapt the implementation of the solution to new versions of the .NET framework.

Table 5.1 shows the comparison of the approaches. The table only shows the approaches that were found to be possible solutions to implement. For example, modifying the .NET compilers and using dummy assemblies are not in the table, since both approaches were found to be impossible to implement.

Signature modification approach	TR	LD	MA	FC	Remarks
Use existing lanaguage construction	–	–	+	+	None
Modify source code with external tool	+	–	–	–	None
Modify an existing compiler	+	–	–	–	Compiler would be Mono
Create a new compiler	+	–	–	–	None
Dummies (retrieval with exernal tool)	+	–	–	–	None
Dummies (retrieval with Visual Studio)	+	+	+	o	Introduces dependency on VS
Dummies (generation with exernal tool)	+	–	–	–	None
Dummies (generation with CodeDOM)	+	+	+	o	None

Table 5.1: Comparison of the found approaches

From the comparison, it is clear that there is one solution that stands out: dummy files generation with Visual Studio and CodeDOM. This solution generates dummies by extracting the type information with Visual Studio and generating the dummy sources with CodeDOM.

There are a number of advantages to this solution. First, the solution is transparent: no modifications of the source code are needed and both steps can be added to the existing compilation process. Second, both the use of Visual Studio and CodeDOM are language independent; one implementation will suffice for all supported programming languages. This and the fact that there are few modification expected by Microsoft to the Visual Studio and CodeDOM interface makes maintainability easier and makes this solution reasonably forwards compatible.

The approach does introduce a dependency on Visual Studio, making the approach less then ideal. However, this disadvantage is relatively small, considering the advantages of this approach and the fact that developers will most likely be using Visual Studio for development with Compose⋆.NET. Therefore, this is the solution that should be implemented as part of the new compilation process.

5.5 The new compilation process

A new compilation process has been conceived that implements the solutions chosen after the comparions of all the found solutions. The new process turns out to be very similar to the old process, but uses a number of different techniques for the same steps.

Figure 5.1 shows an overview of the new compilation process. We will describe the major components of the process in more detail:

Visual Studio The first step in the compilation process remains extraction of the necessary type information from the original source files. However, instead of a code parser, we use Visual Studio to analyze the source files. The extracted type information is still represented in a language model, which allows us to represent the source code in a language independent manner.

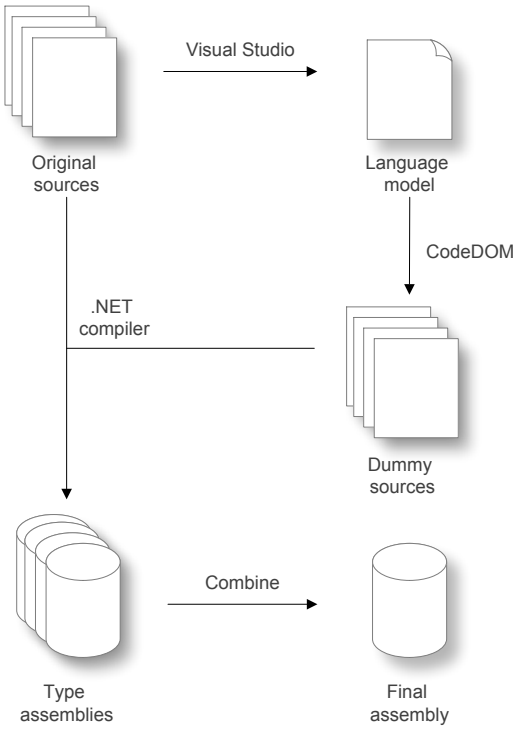


Figure 5.1: Overview of the new compilation process

CodeDOM The second step is the dummy source generation, which generates a dummy version of each of the original source code files. The required source code information is taken from the language model and converted to a representation in CodeDOM. The functionality provided by CodeDOM is then used to create the dummy source files. By using dummy source files instead of dummy assembly, we keep this solution compatible with .NET 2.0 compilers.

.NET compiler The .NET compilers are now used in only one step; the compilation of the orginal source files. The original source files are compiled one at a time. Each file is compiled with the dummy source files, which provide the type information otherwise provided by the other original source files. Because the type information in the dummy source files has the signature modifications applied, signature modification is taken into account at compile-time.

There is a dummy version of the original source file that is to be compiled. This dummy version contains the same type information, which leads to compiler errors. To prevent errors, the dummy version of the original source file is left out of the collection of dummy source files.

Combine assemblies This step has not changed from the previous solution. Because the original source files need to be compiled one at a time, the compilation step results in a number of assemblies. Each of these assemblies contains the fully implemented types of one of the original source files and the dummy types of the other source files. These assemblies need to be combined into one fully implemented assembly.

5.6 Evaluation

To determine whether the new compilation process is an improvement over the old process, we evaluate the new process against the problem that were found in the old process and describe any problems introduced by the new compilation process:

Compatible with .NET 2.0 The compatibility problem in the old compilation process was caused by the use of a dummy assembly. The new process no longer uses a dummy assembly, but has replaced this with the dummy source files. Without the use of dummy assemblies, the new process should be compatible with the .NET 2.0 compilers.

Language dependencies The language dependencies in the old process were caused by the tools used to analyze the original source code and the generation of the dummy source files. These tools have been replaced in the new process by Visual Studio for the analysiss and CodeDOM for the dummies generation. This should remove any language dependencies, since both Visual Studio and CodeDOM represent source code in a language independent manner.

Self-referencing types The problem with self-referencing types is caused by the fact that during the compilation of a self-referencing type, the type information from the containing source file is used. The new compilation process still uses the unmodified source files during compilation and the compiler will still use the unmodified type signature. This means that self-referencing types are still not supported.

Introduced dependencies The new compilation process introduces one disadvantage: a dependency on Visual Studio and CodeDOM. Because CodeDOM is part of the .NET framework, it should not pose a big problem. The dependency on Visual Studio requires Visual Studio to be installed for Compose⋆.NET to function. This could be considered a relatively small disadvantage, considering the advantages of using Visual Studio and the fact that developers will be likely to use Visual Studio for developing with Compose⋆.NET.

Chapter 6

Implementation of the new solution

This chapter documents the changes made to Compose⋆/.NET to implement the new approach. The added or modified functionality should enable signature modification through composition filters for Compose⋆/.NET.

First, we look at the general structure of the Compose⋆ tool as it applies to both the previous and the modified implementation. We then look at the previous implementation, before looking at the modified implementation and all the changes that have been made. The two new tasks that are introduced in the new implementation, are described in more detail in Section 6.3.1 and Section 6.3.3. Finally, we evaluate the modified implementation and compare it with the theoretical solution described in the previous chapter.

6.1 General structure

Both the previous and improved implementation of Compose⋆/.NET use the *Microsoft Build Engine* (MSBuild) to build Compose⋆/.NET projects. The main functionality of Compose⋆/.NET is implemented in tasks that can be executed by MSBuild. To store and exchange data between the separate tasks, a data repository is used. These three main parts that make up each Compose⋆/.NET will be described in this section.

6.1.1 MSBuild

The *Microsoft Build Engine*, or MSBuild, was introduced by Microsoft with the release of the .NET 2.0 framework. MSBuild is designed to be the new build platform for all Microsoft development. For example, Visual Studio 2005, which was introduced with .NET 2.0, depends on MSBuild for the building of its projects.

Since MSBuild is part of the .NET 2.0 framework, it can be used to compile projects without the need for a separate development platform. The MSBuild system is an executable (msbuild.exe) that executes a number of actions described in one or more

XML-based project files. These project files contain all information needed for building the project.

The five main elements defined in a build file are:

Project; the root element of any project build file.

Targets; these are the entry points of any build process. Targets define which tasks will be called during the build process. Examples of common targets are building, rebuilding or cleaning a project.

Tasks; these are the parts of the build file that are to be executed. Besides the provided tasks, such as copying or deleting a file, it is possible to create and add custom tasks. More on these in tasks in Section 6.1.2.

Properties; properties can be used to define and store values that will be used during the build process. These can contain properties of the project, such as debug settings, but custom properties can be defined.

Items; these contain the files that are part of the project and the type of file. Examples of items are the source code files and resource files.

6.1.2 MSBuild tasks

As was already mentioned, MSBuild calls certain tasks to perform the actions demanded by the build file. A large number of standard tasks are provided and can be used in project files. These tasks provide functionality for file and directory operations, executing other programs (such as the C# compiler) and for displaying feedback.

Besides the provided common tasks, it is possible to create custom tasks. Microsoft provides functionality to program custom tasks that can be called by MSBuild. A custom task can be created as a .NET task that either inherits from the `Task` class or that implements the `ITask` interface directly. The `ITask` interface provides an interface which allows MSBuild to execute the custom task from a build file.

A large part of the functionality of Compose★ is implemented in a number of these tasks and the difference between the old implementation and the new implementation can be found in a number of modified custom tasks. The specific tasks that make up Compose★, will be described in more details in Section 6.2 (the old implementation) and Section 6.3 (the new implementation).

6.1.3 Data repository

A central location for all Compose★ settings was needed to allow all separate parts of Compose★ to be able to access the same settings. All settings are therefore stored in a single file that can be accessed by any Compose★ part that requires it. This single file is the data repository.

The data repository has an XML notation to store the different settings. To provide an easy way to access and modify the settings in the file, we use serialization. Normally,

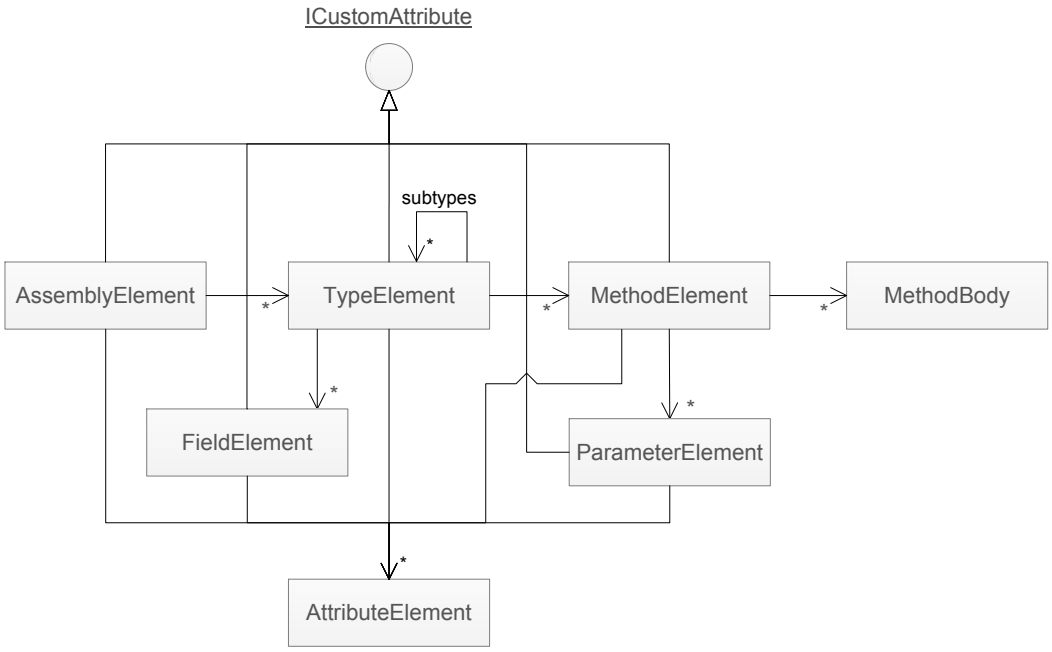


Figure 6.1: Language model class diagram

the settings can be accesssed through an instance of the class that contains the settings and provides methods to access and modify them. The state of this settings object is then saved to disk by serializing the object to an XML file. This XML now stores the state of the object at the time of serialization.

Loading the settings involves deserializing the XML file back into a settings object. The state of the new settings object will be the same as the state of the previous settings object at the time of serialization. This technique allows to easily load and save settings, while at the same time provide an easy interface to interact with the settings within source code.

The use of serialization makes it possible to use the settings file regardless of programming language. Compose has components that are written in C# and Java, which both support serializiation. The settings file can therefore be used to transfer settings between different parts of Compose⋆, even if these parts are programmed in different languages.

Since many parts of Compose⋆ use the settings file and expect the current structure of the settings file and classes, we must make sure that the data repository remains backwards compatible. Breaking the backwards compatibility could require the modification of large parts of Compose⋆ to handle the new layout of the repository.

Figure 6.1 shows the class diagram of the language model. This language model is part of the repository and can be used to represent assemblies, types, method, etc. in a language independent manner. The language model in the repository is used to

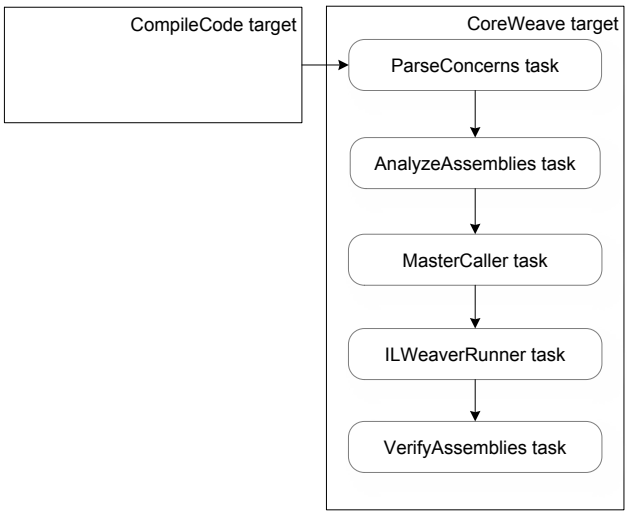


Figure 6.2: The tasks diagram of the old implementation

represent the types found in the original source code files.

6.2 Previous implementation

It is important to note that the previous implementation of Compose★/.NET does not support signature modification. With no support for signature modification, we can use the standard .NET compiler to compile the code. This means that a large part of the code compilation is very similar to the code compilation in a standard .NET project file.

Figure 6.2 shows the order in which the targets and tasks are called. We will now look at the different targets and tasks in a bit more detail to see what each does. We will describe each of the tasks in the order they are executed by MSBuild.

CompileCode target The CompileCode target is executed before the main Compose★ tasks. It is the standard target used by MSBuild to compile the code in the project. This target is responsible for the compilation of the source code to the final assembly. The CompileCode target is executed before the CoreWeave target that contains all the tasks described in the diagram.

One modification in the CompileCode target is the execution of the ParseConcerns task before compilation to make sure that any code in the concerns is compiled too.

ParseConcerns task The ParseConcerns task is responsible for the parsing of the concern files in the project. A custom made parser is used to parse the concern syntax.

Any syntax errors found will be displayed in the output and will stop the execution of the project build file.

The task is also responsible for the extraction of any code that is embedded in the concern files. The extracted code will be saved into separate source files. This enables the embedded concern code to be included in the compilation by the standard .NET compilers.

The final responsibility of the task is the creation of the data repository. The information parsed from the concern files is then stored in the data repository.

AnalyzeAssemblies task The AnalyzeAssemblies task analyzes the assemblies used in the project. These assemblies are either the result of the compilation by the .NET compiler or the assemblies that are referenced by the project. The task analyzes the types in the created assembly and the types that are referenced from the created assemblies. Analyzing assemblies is independent of any programming language, because the assemblies contain the data in CIL. Mono.Cecil is used for the analysis of the assemblies.

The result of the analysis of the types in the created assemblies is stored in the data repository as an instance of the language model that is part of the repository defininition. The modified repository is saved to disk.

MasterCaller task The MasterCaller is responsible for executing the MASTER component of Compose⋆. The MASTER component is present in the Composestar.jar. This part of Compose⋆ is written in Java rather than C#. This means that the MASTER has to be executed indirectly. The MasterCaller task executes the jar file by creating a new process in which Java runs the Composestar.jar. The task captures any output by the process and displays this to the developer.

The MASTER component is responsible for the analysis of the concerns, the order in which they have to be applied and creating the weave specification.

ILWeaverRunner task The ILWeaverRunner task takes the assemblies generated by the compiler, the concerns information generated through the MasterCaller task and applies the concerns to the code in the assemblies.

All concerns are applied to the assembly by weaving IL code that represents the concern code, into the targeted method already present in the assembly. This inlining of the concerns removes the need for a runtime part of Compose⋆/.NET and increases performance of the generated code.

VerifyAssemblies task The VerifyAssemblies task checks the validity of the generated assemblies. This is done by calling the *Microsoft .NET Framework PE Verifier* tool. This tool can verify the PE structure, the metadata and the IL code in the assembly. The tool is simply called from the task and any output is caught and displayed.

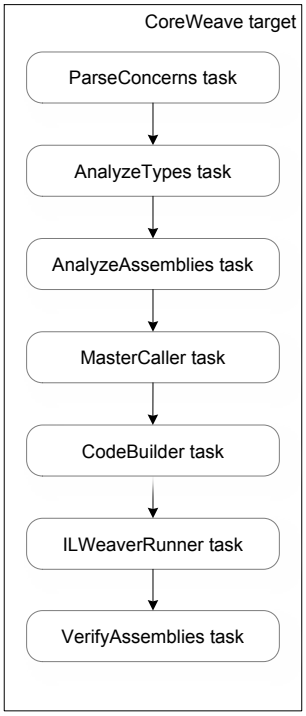


Figure 6.3: The tasks diagram of the new implementation

6.3 Improved implementation

The improved implementation is designed to add support for signature modification, while at the same time remaining backwards compatible with the old implementation. The backwards compatibility allows us to modify the existing tasks as little as possible. Rather, two new tasks have been implemented to replace old tasks in a transparent manner.

The signature modification requirement means that it is impossible to keep the CompileCode target for the compilation of the source files. The CompileCode target uses the standard .NET compilers to compile the original source files. This means that the CompileCode target does not provide support for signature modification, because the compilers do not take composition filters into account, it.

Therefore, the CompileCode target is replaced by two new tasks that will be responsible for the applying of signature modification to the source code and compiling the source code to the target assemblies.

Figure 6.3 shows the new order in which the tasks are executed. The tasks in the project build file will be described in the order they are executed by MSBuild. After the new project build file is described, we will go into more detail of the two new tasks: the

AnalyzeTypes task and the CodeBuilder task.

ParseConcerns task The ParseConcerns task remains unchanged. All functionality for signature modification in the concern files is already present and the filters that expand or reduce signatures can be parsed and stored in the data repository.

AnalyzeTypes task The AnalyzeTypes task is the first of the two new tasks. It is responsible for the analysis of the source code files in the project and thus replaces part of the CodeCompile target and the AnalyzeAssemblies task.

The task analyzes the types in the source code files that are part of the project. This is done through Visual Studio which helps us to analyze the source code in a language independent manner. The found type information is stored in the data repository as an instance of the language model.

We will describe the functionality of the AnalyzeTypes task in more detail in Section 6.3.1.

AnalyzeAssemblies task The AnalyzeAssemblies task had to be modified, since it relied on the compiled final assembly. This assembly is not yet available at the time of the execution of the AnalyzeAssemblies task and therefore cannot be analyzed. The analysis of the created assemblies is removed and only referenced assemblies are now analyzed.

MasterCaller task The MasterCaller task itself does not need any modifications to support signature modification, since it is only responsible for the execution of the main component of Compose⋆/.NET; the MASTER component.

One of the responsibilities of the MASTER part of Compose⋆/.NET is to save the types, the methods and the applied filters to disk. These saved types and methods include methods that are added by signature expansion, but do not include methods that are removed by signature reduction. A modification is therefore required to support the recognition of methods that are removed by signature reduction. This data is used by the next task to actually apply the signature modification.

CodeBuilder task The CodeBuilder task is the second of the new tasks and is introduced to replace the CompileCode target. It takes the type info and concerns from the data repository and applies the signature modification to the targeted methods. The task then creates the dummy source files from the signature modified types. The dummy sources are then used for the compilation of the original source files. The resulting dummy assemblies are merged into the final assembly, which can be used by the next tasks.

We will go into more detail of the new CodeBuilder task in Section 6.3.3.

ILWeaverRunner task The ILWeaverRunner task does not need any modifications to support signature modification; since signature modification has already been applied to the final assemblies in which any filter code will be weaved.

VerifyAssemblies task The VerifyAssemblies task does not require any modification for the support of signature modification.

Modifications to the data repository

The language model in the data repository has been expanded to include a number of language constructions that can be found in a number of programming languages (most notably C# 2.0). The important additons are:

- events; events are represented as fields in the language model.
- delegates; delegates are represented as methods in the language model.
- properties; properties too are represented as methods in the language model.
- generics; it is possible to indicate in the language model whether a type, method or field is generic.

Most of these additions are represented as elements that were already in the language model. This has been done to keep the updated language model backwards compatible. For example, by representing properties as methods, other parts of Compose⋆/.NET can still reason about properties even though there is no explicit support for properties.

The support for generics in the language model is also backwards compatible. However, the support for generics in the language model does not mean that generics are fully supported by Compose⋆/.NET. Generics are supported during source parsing and dummy generation, but there is currently no full support for generics in Compose⋆/.NET.

Furthermore, a number of compiler settings can now be stored in the data repository. These settings are retrieved from Visual Studio and saved by the AnalyzeTypes. The settings are used in the new CodeBuilder task when compiling the original source files.

6.3.1 Code analysis

The main responsibility of the AnalyzeTypes task is to analyze the source code files in the project in a manner that is as much as possible language independent. This is achieved by no longer using a number of code parsers to analyze the source code, but by using Visual Studio for this task.

Visual Studio provides an interface through which Visual Studio can be controlled and data can be read out. More on this interface for communicating with Visual Studio in Section 6.3.2. Through this one interface, we can analyze all languages supported by Visual Studio. This gives us a method to analyze Visual C#, Visual J#, Visual Basic and Visual C++ without having to use a code parser for each language.

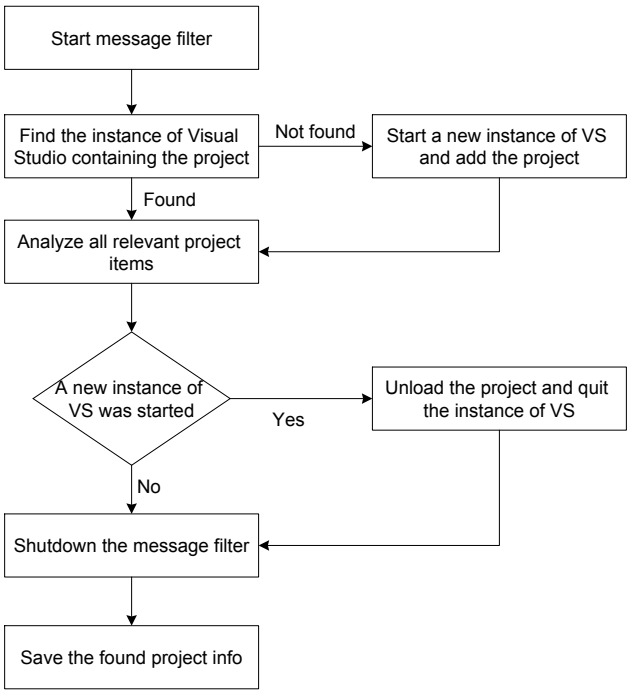


Figure 6.4: Flow chart of the type analyzing process

The first step in the CodeAnalyzer task is the starting of the message filter. This filter is responsible for the proper communication between the task and Visual Studio. More information on the message fiters can be found in Section 6.3.2.

The next step is to find an instance of Visual Studio that contains the project we want to analyze. We look for all instances of Visual Studio currently running. For each found Visual Studio instance, we look at the loaded projects. If one of the loaded projects is the project we want to analyze, then we will use the reference to this instance of Visual Studio and the loaded project.

If there are no instances of Visual Studio running or none of the running instances have the target project loaded, then we start a new instance of Visual Studio. If a new instance is successfully started, then our project is added to the default solution of the new instance.

We can now use Visual Studio to analyze all items present in the loaded project. The first items we want to analyze are the project settings. These include build/compilation settings and the external assemblies referenced by the project.

The main target of the project items we want to analyze are the source code file elements. Besides the properties of the files themselves, we are interested in the code elements contained in the files. These code elements are analyzed and are represented as parts of the language model. This leads to an instance of the language model that contains a representation of all the source code that was found in the project.

After all code elements have been analyzed successfully, the instance of Visual Studio

is closed if it was created by us. The message filter is shut down after all communication with Visual Studio has stopped. Finally, the new instance of the language model is added to the data repository.

If the code analysis fails at any point, then the CodeAnalyzer task will fail and the build process will be halted. Examples of failures are: failures during the creation of a new instance of Visual Studio and any unexpected exceptions thrown by Visual Studio. The reason for the halting of the analyzing process will be displayed.

6.3.2 Visual Studio automation

The functionality of Visual Studio automation is provided in one namespace; EnvDTE. EnvDTE is an assembly-wrapped COM library containing the objects and members for Visual Studio core automation [39]. The EnvDTE namespace contains the types to represent and manipulate parts of Visual Studio. These parts include windows, solutions, projects and code elements.

The library allows us to communicate with an instance of Visual Studio through the COM interfaces exposed by Visual Studio. COM (*Common Object Model*) provides a way to communicate between separate components in a fashion that is independent of programming language, implementation and location. COM is one of the main techniques in Microsoft Windows for communication between separate processes or applications.

New functionality exposed by a new version of Visual Studio can be found in a separate namespace, leaving the original EnvDTE namespace unchanged. For example, functionality specific to Visual Studio 2005 can be found in the EnvDTE80 namespace. Providing new functionality in new namespaces preserves backwards compatibility and should facilitate code migration. This allows code that relies on the old library to function on newer versions of Visual Studio.

Figure 6.5 shows the parts of the EnvDTE namespace we are interested in. The DTE object is the top-level object (DTE being the top-level object within EnvDTE and DTE2 within EnvDTE80). The Solution and Project classes represent the solution and any projects that are loaded in Visual Studio.

The `CodeElement` object represents a code element within the code of a source file in Visual Studio. This object can be cast to other objects, such as `CodeClass`, to represent more specific source code elements. Languages that are represented by this model are Visual Basic, Visual C#, Visual J# and Visual C++(.NET). Visual C++(.NET) is supported by the Visual Studio code model, but to analyze Visual C++-specific code, there is an extension to the core code model. A complete chart of the Automation Object Model can be found at [37].

Although the EnvDTE namespace makes it appear that Visual Studio can be controlled through a number of types and methods, the communication is actually based on messages that are sent back and forth. These messages are normally completely transparent when controlling Visual Studio, but they can be the cause of exceptions in the calling program.

There are two error messages that occur regularly: "Application is busy" and "Callee was rejected by caller." Both of these messages are thrown when Visual Studio indicates

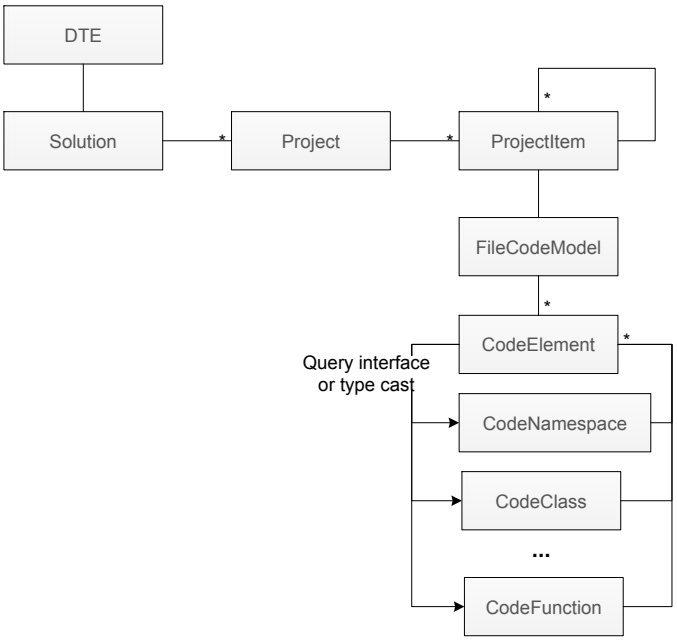


Figure 6.5: Part of the Visual Studio object model class diagram

it is not ready to execute the command. These messages halt the normal execution of our task since they result in an exception being thrown.

These exceptions need to be caught and the original commands have to be resent to ensure the program (in our case: the AnalyzesTypes task) can continue, but catching these in the source code is very impractical since the exception can be thrown with any command issued to Visual Studio.

A message filter is introduced to filter out the returned "Application is busy" and "Callee was rejected by caller" messages that cause the exceptions. The message filter is placed between the program (the AnalyzesTypes task) and Visual Studio and works transparently for both the program and Visual Studio.

If a sent command causes Visual Studio to return one of the two messages, then the returning message is intercepted by the message filter and the original command is resent. If the returning message indicates the command has been executed successfully or contains a different error, then the the returning message simply passes through the message filter. For more information on the message filter, see [41].

It could be possible to use the source code representation provided by the classes in the EnvDTE namespace, to represent the found types throughout Compose⋆/.NET. This would then replace the current language model in the data repository, removing the need for the conversion currently performed.

There are two problems with replacing the current language model. The first problem is that the classes in the EnvDTE do not support serialization. This support either has to be added or another solution has to be found for the data repository. The second

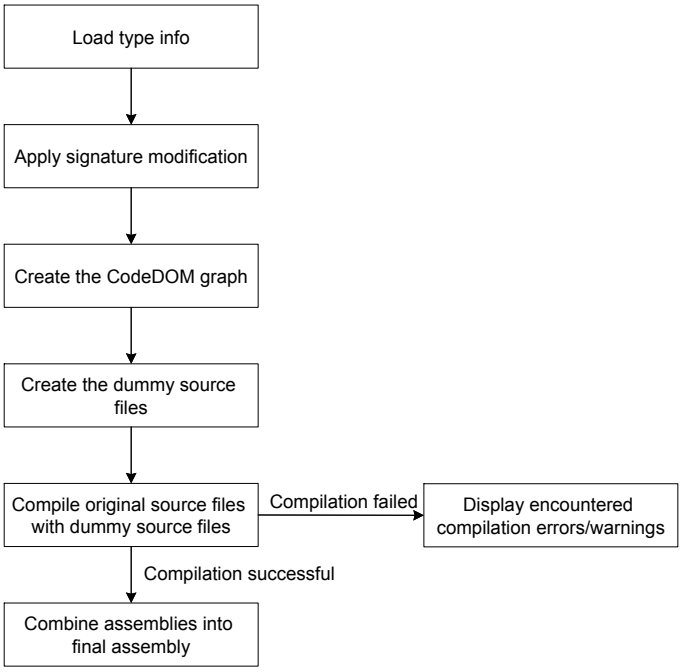


Figure 6.6: Flow chart of the code building process

problem is that replacing the language model would break the backwards compatibility of the new implementation. Every component of Compose⋆/.NET that uses the language model would have to modified to work with the new representation.

6.3.3 Code compilation

The *BuildCode* task is responsible for applying the signature modification, the creation of the dummy sources, the compilation of the original sources and creating the final assembly. Figure 6.6 shows the components of the code building process and the order in which they are executed. The components will be described below in more detail in the same order.

Load project and type information. The first step of the process is to load all necessary information from the data repository. The most important information for the code building is the type information. Other information needed are the compiler settings of the project. Besides the info from the data repository, the info from the weave specification file is loaded.

Apply signature modification. The next step of the process applies the signature modification to the types loaded from the data repository. Any signature modification is determined in the previous task (the *MASTER* task) and is described in the weave

specification, along with all other functionality that is introduced by composition filters. We use the type information and the weave specification for both signature expansion and signature reduction.

Applying signature expansion involves adding a method to the targeted type. Types and methods in the weave specification are looked up in the type information. If the type is found, then we look for the targeted method. If the method is not found and the applied filter is a dispatch filter, then the method is added to the type in the type information.

We apply signature reduction by removing an existing method from a type. We look up the targeted type and method in the type information. If the type and method are found and the weave specification indicates that an unconditional error filter is applied to the method, then the method is removed from the type in the type information.

Create a CodeDOM graph. After the signature modification is applied to the types in the type information, we convert all type information to the corresponding CodeDOM classes. By creating a CodeDOM graph with these CodeDOM classes, we can represent the dummy source files in CodeDOM. CodeDOM and the classes used in compilation, will be described in more detail in Section 6.3.4.

Generate the dummy source files. This step takes the CodeDOM tree created in the previous step, and generates the dummy source files from it. The functionality to generate source code from a CodeDOM tree is provided in the .NET framework for all supported programming languages. This allows us to generate source code files in the same programming languages as the original source files, which is a requirement for the next step.

A dummy source file will be generated for each original source file in the project. The generated dummy source file will contain dummy versions of the types found in the orginal source file. These dummy types are types with the proper signature but no implementation. The dummy source files can be found in the *Dummies* folder that is created in the *StarLight* folder.

Compile the original source files. In this step, we take the original source files and compile them with the dummy source files. Each original source file is compiled separately with all the dummy source files, accept for the dummy source files containing types that are defined in the original souce files. This allows the original source file to be compiled against the signature modified versions of the other types.

Because each original source file is compiled separately, we end up with multiple dummy assemblies. Each created dummy assembly has the same name as the original source file used in the compilation. The dummy assemblies can be found in the same *Dummies* folder as the dummy source files.

The compilation functionality is provided by the same `CodeProvider` classes that are used for the dummy source generation. These classes allow us to write source code that

can use the .NET compilers to compile the source files. Any feedback given by the .NET compiler is returned to the BuildCode task.

Should an error be encountered during compilation, then this will halt the Compose⋆/.NET process. Any error and/or warning messages will be returned to the developer in a manner that is similar to any errors or warnings during the compilation of a standard project by a .NET compiler.

Combine the dummy assemblies. The final step consists of combining the dummy assemblies to form the final assembly. Each of the dummy assemblies only contains the functionality of the types that were in the original source file that was compiled to this assembly. The other types in the dummy assembly are the dummy types with the proper type signature, but no implementation.

Therefore, all the fully implemented types in the dummy assemblies must be combined into one assembly that will become the final assembly. First, one of the dummy assemblies is chosen to become the final assembly. We then take the implemented types from all the other dummy assemblies and copy these into the final assembly, replacing the dummy versions of the types.

After the types are all combined into the final assembly, we must recreate all references. These can be references to each other and to external resources, such as types in other assemblies. Finally, if the assembly is an executable then we set the entry point of the assembly to the main method defined in the source code. If necessary, we rename the final assembly to the name of the output assembly as indicated in the project settings.

Mono.Cecil is used for the combining of the types and fixing the final assembly.

6.3.4 CodeDOM

The CodeDOM provides types that represent many common types of source code elements. You can design a program that builds a source code model using CodeDOM elements to assemble an object graph. This object graph can be rendered as source code using a CodeDOM code generator for a supported programming language. The CodeDOM can also be used to compile source code into a binary assembly [38].

Figure 6.7 shows a part of the `System.CodeDom` namespace. The `CodeCompileUnit` is the root of any CodeDOM graph and represents an assembly of module. Most other objects are self-explanatory and represent code elements, such as namespaces, classes, methods, etc. More information on the `System.CodeDom` namespace can be found at [40].

The `CodeDomProvider` class can be found in the `System.CodeDOM.Compiler` namespace. This class serves as a base class for all `CodeDomProvider` implementations. These implementations provide functionality for the generation of source code from a CodeDOM graph or for compiling the CodeDOM graph to a binary assembly.

These implementations use three classes for the code generation or compilation of a CodeDOM graph. The most important class is the `CodeCompileUnit` that contains that CodeDOM graph. The `CompilerParameters` class represents the parameters to be used by the compiler. Finally, the results of the compilation are returned in a `CompilerResults` object.

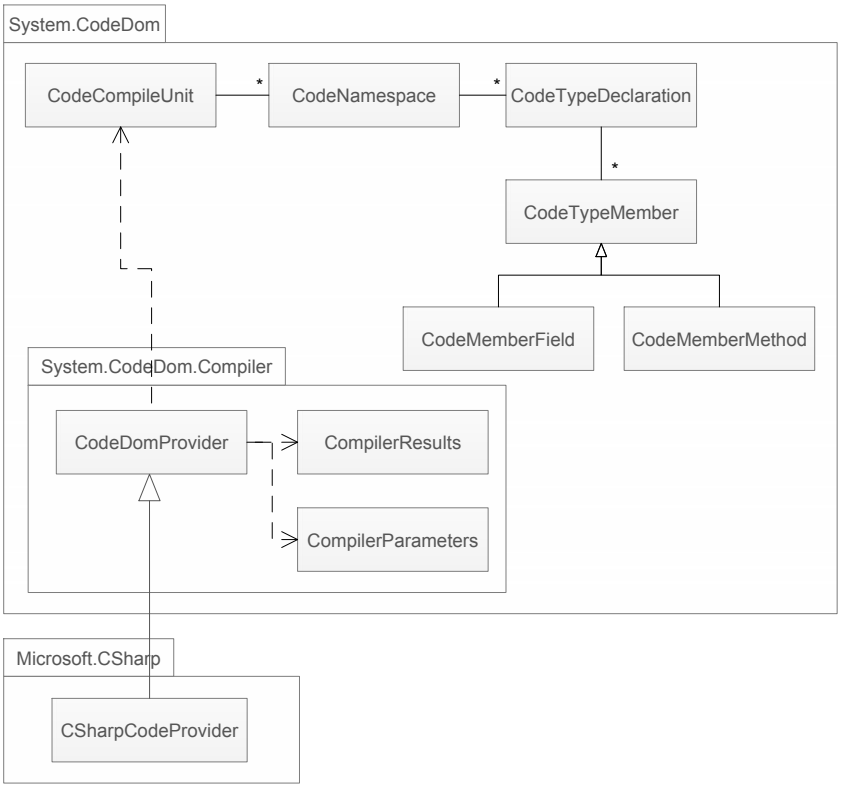


Figure 6.7: class diagram of a part of the CodeDOM

Figure 6.7 also shows an example of a class that implements the `CodeDomProvider` class. The `CSharpCodeProvider` class implements `CodeDomProvider` to provide functionality for generating source code in C# and compiling a CodeDOM graph to an assembly. The compilation of a CodeDOM graph by the `CSharpCodeProvider` class is actually achieved by generating source code in C# and using the standard C# compiler to compile the generated source code to an assembly.

A case could be made to use CodeDOM for the code representation in Compose⋆/.NET instead of the language model as is decribed in Section 6.3. There are two downsides to using the CodeDOM code representation. First, it would require the modification of large parts of Compose⋆/.NET to use the new code representation. Second, the `CodeCompileUnit` class cannot be serialized to XML, because one of its public properties is an IDictionary type. Both of these downsides are solvable, but the amount of work makes it unpractical.

6.4 Evaluation

If we compare the implementation to the new approach in theory, then we can conclude that it implements the theoretical solution without any major problems. The new implementation introduces support for signature modification while remaining backwards compatible with the old implementation.

The new implementation relies heavily on Visual Studio, but this downside was already identified when looking at the theoretical solution. The same goes for the lack of support for signature modification on self-referencing types.

However, a few minor problems were found during the implementation of the new approach.

Projects. The previous implementation of Compose⋆/.NET uses a custom Visual Studio project type. This Compose⋆/.NET project type was found to be incompatible with the new implementation. The reason for this is found in the AnalyzeTypes task. Visual Studio is unable to analyze the source code files if the loaded project is not recognized as either a Visual C# or J# project type. This is the case for the current Compose⋆/.NET projects.

Two solutions are available to make the Compose⋆/.NET project compatible. The first solution would be to modify the project type to be a subtype of the Visual C# or J# project type. The second solution would be to modify the project to contain a nested project of the Visual C# or J# project type, which can be analyzed.

Language dependencies. Unfortunately, the new implementation is not completely language independent. The one language dependency remaining revolves around the static main method that is the entry point for an executable assembly. Visual Studio does not provide any functionality to recognize these methods during the analysis of the source code files.

Therefore, these methods have to be recognized manually in a language dependent manner. The language dependency comes from the fact that the entry point is different in the different programming languages. For example, the static main method has to be named `Main` in Visual C#, whereas this method is always called `main` in Visual J#.

Debug support. Currently, there is no proper support for creating debug versions of the assembly in the new implementation. The original source files can be compiled into debug versions of the dummy assemblies. These debug versions are then combined to create the final assembly. But although the final assembly now contains code that is suitable for debugging, the required *program database file* (PDB) is not created from the program database files accompanying the dummy assemblies. Support for writing PDB files is available through *Mono.Cecil*, but is currently not implemented.

Chapter 7

Testing the implementation

The new approach has been implemented as described in the previous chapter. The implementation follows the theoretical approach and should be backwards compatible with the previous implmementation while at the same time supporting signature modification. Both the backwards compatibility and the signature modification support require testing.

First, we test the backwards compatibility of the improved implementation. The backwards compatibility requires the new implementation to be able to replace the standard compilation process. The improved compilation process must compile any source code that can be compiled by the standard .NET compilation process.

The second series of tests is intended to test the requirement for support of signature modification. From these tests, we derive a number of problems with the improved implementation and will give a number of recommendations to solve these problems.

7.1 Testing the backwards compatibility

The first category of testing revolves around testing the backwards compatibility of the new implementation. The old implementation used the .NET compilation process to create the assemblies. Therefore, the new implementation must have a compilation process that is equivalent to the default compilation process.

The tests will focus on two parts of the new compilation process. First, we will look at the dummy source files generation. The second part tests the combining of the dummy assemblies in the final assembly. The compilation of the original source files (with the dummy sources) is done with the .NET compilers and therefore does not require testing.

The dummy source files generation involves analyzing the source code with Visual Studio, representing the code in the data repository through an instance of the language model and converting the instance into the dummy source files with CodeDOM. The type in the resulting dummy source code files must have the same signature as the original types.

The creation of the dummy assemblies involves taking the generated dummy source files and compiling the original source files with them. The resulting dummy assemblies, each containing one full type, will have to be combined into the final assembly. This assembly must contain the proper IL code and executing the assembly should give the same behavior as executing the assembly, if it was built with the original compilation process.

Both parts of the new compilation process can be tested with one test suite. By comparing the signatures in the created dummy source files with the original types signatures, we can see if the first part of the compilation process is correct. By comparing the final assembly with an assembly created with the .NET compilation process, we can see if the second part of the implementation works correct.

Microsoft does not provide a test suite for either C# or J#, but a test suite was created to check the compliance of the Mono compiler. This test suite contains a large number of source files that test all parts of the C# specification. Unfortunately, the Mono test suite contains a large number of source files with a type in each file and with each type having a static `Main` method.

This would require every source file to be added to a Compose⋆ project one at a time. Though this is possible, it is not the most practical and time-efficient method. Instead of using the Mono test suite, we have created a test suite to test the language constructions in C#. The files in this test suite can all be compiled by Compose⋆ in a single project.

Unfortunately, no test suite exists for J#. This means that we cannot test the dummy source files generation with J# source files. A test suite for J# will presented as future work, together with expanding and automating the current C# test suite.

Table 7.1 shows the global results for each language construct tested with the test suite. The new implementation is said to be backwards compatible if all tests are passed. There are four test results for each language construct:

Language model (LM) The test for the language model is passed if the language construction is properly represented in the language model found in the data repository.

Dummy generation (DG) The test for dummy generation is passed if the dummy version of the type has the same signature of the original type and the dummy source file contains valid source code.

Code compilation (CC) The test for code compilation is passed if the original source files can be compiled together with the dummy source files to the dummy assemblies.

Valid IL code (IL) The test for valid IL code is passed if the final assembly contains IL code that is equivalent to IL code produced by the standard compilation process.

Execution (EX) The test for execution is passed if the final assembly exhibits the same behavior at runtime as an assembly produced by the standard compilation process.

There are three possible results for every stage of testing. A 'v' indicates that all tests have been passed. A '*' indicates that the test has failed for one or more implementations of the language construction, but not all constructions. Testing can be continued without the failing implementations. A '-' indicates a full failing of the tests. All following tests cannot be performed for this language construction (indicated by 'n/a').

Language construction	LM	DG	CC	IL	EX
Namespaces	v	v	v	v	v
Nested namespaces	v	v	v	v	v
Classes	v	v	v	v	v
Nested classes	v	-	n/a	n/a	n/a
Interfaces	v	v	v	v	v
Structs	v	v	v	v	v
Enumerations	v	v	v	v	v
Inheritance class : class	v	v	v	v	v
Inheritance class : interface	v	v	v	v	v
Fields (access modifiers)	v	v	v	v	v
Fields (types)	v	v	v	v	v
Methods (access modifiers)	v	v	v	v	v
Methods (return types)	v	v	v	v	v
Methods (parameters)	v	v	v	v	v
Properties	v	v	v	v	v
Generics (interfaces)	v	v	v	v	v
Generics (classes)	v	v	v	v	v
Generics (fields)	v	v	v	v	v
Generics (methods)	v	v	v	v	v
Events	v	v	v	v	v
Delegates	*	v	v	v	v
References	v	v	v	v	v

Table 7.1: Results of the backwards compatibility testing

Two problems were encountered during testing with the test suite. The first problem involves global delegates. Delegates are methods that can be part of a type or they can be defined globally. Delegates that are a member of a types pass all tests, but global delegates do not.

There is currently no support for global delegates in the language model. Delegates are represented in the language model as methods. Methods can only be defined as members of a type. A solution could be to allow the `AssemblyElement` to contain methods making it possible to represent global delegates.

The second problem involves nested classes. Nested types are supported by the language model and a nested class was properly analyzed in Visual Studio and represented by the language model. The nested class was not present in the dummy source file, indicating that the failure lies in the generation of the CodeDOM tree. This problem is

fixed by implementing support for nested classes in the CodeDOM generating class.

7.2 Testing the signature modification support

Besides testing the backwards compatibility of the new implementation, we have to test the support for signature modification. The tests should not only test the support for signature modification, but should also test if the behavior of the new implementation during compile-time and runtime is as expected.

We will first look at the scenarios introduced by the signature modification support. From these scenarios, we will derive a number of tests. Since a large number of these tests will give results that are as expected or very similar in behavior, we will not described all these test results. For completeness, all these test results can be found in appendix A. Any test that returns a result that is unexpected or helps us deduce a more general problem will be described in Section 7.2.3.

7.2.1 Scenarios introduced by signature modification

To decide on the tests required for a complete testing of signature modification, we need to find all scenarios that can occur as a result of signature modification. To find all scenarios, we must look at everything that is involved with a signature modification. There are three areas: the signature modification operation itself, the target of the operation and its properties (i.e. a method in a type) and the interaction of the target with other code.

Signature modification introduces four operations on the source code: signature expansion, signature reduction, conditional signature expansion and conditional signature reduction. The static signature expansion and reduction introduce new behavior at compile-time. The conditional signature modifications also introduce new behavior at runtime.

All signature modification operations are performed on methods. Any signature modification involves either adding a new method or removing an existing method. The result of any modification operations can depend on the properties of the targeted method. The keywords for these properties are: `abstract`, `static` and the access modifier (such as `public`).

Finally, the result of an operation on a method depends on the interactions this method has with other methods. There are three possibilities: there is no interaction, the method is part of an inheritance structure or the method is part of an overwriting structure.

If there is no interaction, then the method is a standalone method. It is not part of an inheritance structure, not does it override another method is it overridden. Adding or removing a standalone method does not affect any type but the type containing the method. Only calls to objects of this type are affected by the signature modification.

Listing 8.8 shows an example of an inheritance structure. If the method is part of an inheritance structure, then the method could be inherited from a super-type and/or

could be inherited by a sub-type. Therefore, adding or removing the method can affect other types and calls to objects of other types.

```
namespace Example {
  class SuperClass {
    public virtual void Test() {
    }
  }

  class SubClass : SuperClass {
    public override void Test() {
    }
  }

  class InheritanceExample {
    public static void Main() {
      SuperClass s = new SubClass();
      s.Test(); //calls the Test method in SubClass
    }
  }
}
```

Listing 7.1: Example of inheritance in C#

Listing 7.2 shows an example of a method being hidden by a method in the subtype. If a method is part of an hiding structure, then the method could be hiding a method defined in a super type and/or be hidden by a method in a subtype. This hiding can be explicit (for example, with the keyword `new` in C#) or implicit by simply defining a method already present in a super type or sub type.

In both cases, the hiding method is not connected to the method it is hiding. If we have an object of the subtype cast to the supertype, then calling the hidden method on the object will call the method in the supertype. An example of this can be seen in Listing 7.2: the `s.Test()` calls the —Test— method in the `SuperClass`.

```
namespace Example {
  class SuperClass {
    public void Test() {
    }
  }

  class SubClass : SuperClass {
    public new void Test() {
    }
  }

  class HidingExample {
    public static void Main() {
      SuperClass s = new SubClass();
      s.Test(); //calls the Test() method in SuperClass
    }
  }
```

```
18 }
```

Listing 7.2: Example of an hidden method in C#

By combining the different operations with the different properties of a method and the different ways in which it can interact with other methods/types, we can deduce a large number of scenarios. By testing these scenarios with the Compose⋆/.NET in its current form, we can find where Compose⋆/.NET introduces unexpected or confusing behavior.

7.2.2 Description of tests

All results of the tests will be described in the following form.

Test number: test title
Description:
The scenario we are about to test is described here.
Expected behavior:
Here we describe the behavior we are expecting to see.
Actual behavior:
Here we describe the behavior we are actually seeing.
Desired behavior:
This describes the behavior we would like to see when signature modification is applied by the Compose* tool.

The tests have been divided into five categories:

- Stand-alone method tests.
- Generic type and method tests.
- Inheritance tests.
- Overwriting methods tests.
- Conditional signature modification tests.

All these tests are done with Compose⋆/.NET in Visual Studio with the language of the source code being C#. For a comparison, all tests have been performed without Compose⋆/.NET. The signature modification was applied manually, where possible.

7.2.3 Test results

Stand-alone method scenarios

The tests with stand-alone methods are performed to test the most basic operations and the effect of the properties of a method on these operations. All signature modifications are unconditional.

Test 1: Stand-alone signature expansion

Description:
A stand-alone method is added to a type and the method is called through an object of this type.

Expected behavior:
The compilation is successful and any calls to the method at runtime succeed.

Actual behavior:
The compilation is successful and any calls to the method at runtime succeed.

Desired behavior:
The actual behavior matches the desired behavior.

Test 2: Stand-alone signature reduction

Description:
A stand-alone method is removed from a type and the method is called through an object of this type.

Expected behavior:
The compilation fails with an error that the called method is not defined in the type of the object. No runtime test is possible.

Actual behavior:
The compilation fails with an error that the called method is not defined in the type of the object. No runtime test is possible.

Desired behavior:
The compilation fails with an error that the called method is not defined in the type of the object. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test is possible.

Generic type and method scenarios

The tests with generic types and methods are similar to the stand-alone tests, but are performed on types and methods with generic parameters. These tests should determine whether generic types and method are recognize by Compose⋆/.NET and whether signature modification can be applied. All signature modifications are unconditional.

```
1  namespace Example {
2    class GenericTypeTest<T> {
3      T _data;
4
5      public void SetData(T t) {
6        _data = t;
7      }
8    }
9
10   class GenericMethodTest {
11     public static void Swap<T>(ref T x, ref T y) {
12       T temp = x;
13       x = y;
14       y = temp;
15     }
16   }
17
18   class GenericsExample {
19     public static void Main() {
20       GenericTypeTest<int> g = new GenericTypeTest<int>();
21       g.SetData(0);
22
23       int x = 0;
24       int y = 1;
25       GenericMethodTest::Swap<int>(0, 1);
26     }
27   }
28 }
```

Listing 7.3: Example of a generic type and generic method in C#

Listing 7.3 describes the source code that is used to test the signature modification of generic types. The `GenericTypeTest` class is used to test signature modification of generic types with the `SetData` method. The `GenericMethodTest` class is used to test signature modification of a non-generic type with the generic method `Swap`.

Test 3: Signature expansion of a generic type

Description:
See Listing 7.3: the `SetData` method is added to the generic `GenericTypeTest` type. The method is called through an object of this type as is defined in `GenericsExample`.

Expected behavior:
The compilation is successful and any calls to the method at runtime succeed.

Actual behavior:
The compilation is successful and any calls to the method at runtime succeed.

Desired behavior:
The actual behavior matches the desired behavior.

Test 4: Signature expansion of a generic type with a different generic method

Description:
See Listing 7.3: the `SetData` method is added to the generic `GenericTypeTest` type, but the parameter of `SetData` is not `T` but `U`. The method is called through an object of this type as is defined in `GenericsExample`.

Expected behavior:
The compilation fails with the message that the `U` type or namespace name could not be found. No runtime test is possible.

Actual behavior:
The compilation fails with the message that the `U` type or namespace name could not be found. No runtime test is possible.

Desired behavior:
The compilation fails with the message that the `U` type or namespace name could not be found. This error message to the developer should be extended to include information that the method is added by a filter. No runtime test is possible.

Methods that are part of inheritance

The tests for methods that are part of an inheritance structure use the code as in Listing 8.8. The `Test` method is either added to or removed from the code through signature modification. These tests should tell us what happens when a composition filter changes an inheritance structure or what happens with calls to the modified structure.

Test 5: Signature expansion of a superclass with an inherited method

Description:
The `Test` method is not present in the signature of both classes. The `Test` method is then added to `SuperClass` and the method is called through an object of `SubClass` that should inherit the method.

Expected behavior:
The compilation is successful. The method called through the object is the method defined in the super type.

Actual behavior:
The compilation is successful. The method called through the object is the method defined in the super type.

Desired behavior:
The actual behavior matches the desired behavior.

Hiding methods

The tests with hiding methods use code as displayed in Listing 7.2. The hiding is done explicitly (through the `new` keyword) and implicitly. Implicit hiding can lead to different behavior (since this also leads to different behavior in the standard C# compilations process).

Test 6: Signature expansion of a subclass with an hiding method

Description:
The `Test` method is present in the signature of `SuperClass`. The `Test` method is then added to `SubClass` hiding the method in `SuperClass`. The method is called through an object of `SubClass`.

Expected behavior:
The compilation succeeds and at runtime the method call calls the `Test` method defined in `SubClass`.

Actual behavior:
The compilation succeeds and at runtime the method call calls the `Test` method defined in `SubClass`.

Desired behavior:
The actual behavior matches the desired behavior.

Test 7: Signature expansion of a subclass with an implicitly hiding method

Description:
The `Test` method is present in the signature of `SuperClass`. The `Test` method is then added to `SubClass`, implicitly hiding the method in `SuperClass`. The method is called through an object of `SubClass`.

Expected behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SubClass`.

Actual behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SubClass`.

Desired behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. This warning message to the developer should be extended to include information that the hiding method is added by a filter. At runtime, the method call calls the method defined in `SubClass`.

Conditional signature modification

Methods added or removed through conditional signature modification are actually always added or never removed. This makes sure that the resulting code can actually be compiled by the standard .NET compilers. Unfortunately, it also means that any errors will occur at runtime after the method has been called, rather then being caught at compile-time.

In an ideal situation we would check the validity of each call to a conditional method at compile-time. This would involve checking the validity of a call with regard to the condition that defines whether the call is valid or not. The condition should always be true at the time of the call to the method.

Unfortunately, this is not realizable and the validity of each call cannot be checked at compile-time. An example in the normal .NET compilers is that they do not accept the conditional initialization of an object, since the compiler cannot check whether the object will always be initialized.

Since the complete type checking of the conditionally added or removed is impossible at compile-time, we will not enter this as desired behavior. Rather, the desired behavior will describe the best way to handle any errors at runtime.

Test 8: Conditional signature expansion

Description:
A stand-alone method is conditionally added to a class. The method is then called twice at runtime; once when the condition is true and the method should be present and once when the condition is false.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. Any calls to the method, when it is added, should just run the code added by the method. Any calls to the method when it is not added, should result in an illegal method call error.

Actual behavior:
The compilation fails with errors by the SIGN part of Compose*. The error message states that the type has an infinite signature.

Desired behavior:
A warning message should be issued to the developer informing that the resulting program might fail at runtime due to the missing method. The error message that is shown when the program fails, should explain that the failure is due to a conditional addition of a method by a composition filter.

Test 9: Conditional signature reduction

Description:
A stand-alone method is conditionally removed from a class. The method is then called twice at runtime; once when the condition is true and the method should be removed and once when the condition is false.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. Any calls to the method when it is removed, should result in an illegal method call error. Any calls to the method when it is not removed, should just run the code added by the method.

Actual behavior:
The compilation fails, since the method is removed permanently at compile-time by Compose⋆ and any calls to the method will be recognized by the .NET compiler as being invalid. Compose⋆ in its current form does seem to recognized conditional signature reduction.

Desired behavior:
The compilation succeeds, but a warning message should be issued to the developer informing that the resulting program might fail at runtime due to the missing method. The error message that is shown when the program fails, should explain that the failure is due to an invalid call to a method that has been removed by a composition filter.

Test 10: Conditional removal of a virtual method from a supertype

Description:
See Listing 8.8: the `Test` method is present as a virtual in the `SuperClass` and is overridden in `SubClass`. The `Test` method is then conditionally removed from `SuperClass`. A `SubClass` object is cast to `SuperClass` and the `Test` method is called on the `SuperClass` object twice; once with the method in place and once with the method removed.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. At runtime, the framework will simply call the `Test` method as defined in `SubClass` since it recognizes that the `SuperClass` object is a cast down `SubClass` object. The conditional removal of the `Test` method from the signature of `SuperClass` will have no effect on the call.

Actual behavior:
The compilation fails, since the method is removed permanently at compile-time by Compose* and any calls to the method on a `SuperClass` object will be recognized by the .NET compiler as being invalid.

Desired behavior:
The compilation succeeds, but a warning message should be issued to the developer informing that the resulting program might fail at runtime due to the missing method. The call to the removed method should fail at runtime, even if the call is actually to the existing method in the subtype. The method was removed from the signature of the object's type and any calls to it should therefore fail. The error message that is shown when the program fails, should explain that the failure is due to an invalid call to a method that has been removed by a composition filter.

Test 11: Conditional removal of a method that is inherited from an abstract supertype

Description:
This is based on the code in Listing 8.8. The `Test` method is present in the `SuperClass` which is defined as abstract, and is implemented in `SubClass`. The `Test` method is then conditionally removed from `SuperClass`. A `SubClass` object is cast to `SuperClass` and the `Test` method is called on the `SuperClass` object twice; once with the method in place and once with the method removed.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. At runtime, the framework will simply call the `Test` method as defined in `SubClass` since it recognizes that the `SuperClass` object is a cast down `SubClass` object. The conditional removal of the `Test` method from the signature of `SuperClass` will have no effect on the call.

Actual behavior:
The compilation fails, since the method is removed permanently at compile-time by Compose⋆ and the compilation will fail with the message that `SubClass` does not implement all methods in `SuperClass`.

Desired behavior:
The problem that occurs is that when the method is removed at runtime from `SubClass`, the `SubClass` actually stops being a valid subtype of `SuperClass`. To prevent this from occurring, it should be impossible to apply conditional signature reduction on methods that implement an abstract method. This check should be applies at compile-time and compilation should fail with an error, informing the developer of the problem.

7.3 Derived general problems

A number of general problems can be derived from the results of the different tests. The biggest problem that occurred during testing was the conditional signature modification not being implemented. This means that any remarks about the runtime behavior of the conditional signature modification tests are theoretical. The conditional signature modification problems that are described here are valid, since they can be deduced from the behavior of .NET itself at runtime.

Compose⋆ introduces no unexpected behavior at compile-time. However, tests (such as test one and test five) have shown that the compiler can display a number of error and warning messages as a result of code introduced by a filter. These messages need to be extended to include information about the filter that introduces that code and the location of the filter and/or code.

No unexpected behavior is introduced by conditional signature modification at compile-time either, since methods that are conditionally added or removed are always

present in the code at compile-time. This does mean that there is no way to check the validity of each call to a conditional method. Compose⋆ should therefore always warn the developer in these cases that runtime exceptions are a possibility.

The static signature modification is all performed and validated at compile-time and should not introduce any unexpected behavior at runtime. However, conditional signature modification can give errors at runtime.

The first problem is the conditional removal of a virtual method from a supertype (shown in test eight). If this method is inherited by a subtype, then it is still present in sub type, but no longer in super type. If we cast an object of the subtype to the supertype and call the method then this should fail, since the method is no longer part of the signature of the supertype. However, the call will succeed because it will actually call the method in the subtype directly, so there is no check for the validity of the call.

The second problem is the conditional removal of a method that is inherited from an interface or an abstract class (shown in test nine). Since the method is not actually removed from the code, the code is valid both at compile-time and at runtime. At the same time, removing the method from the subtype could be seen as breaking subtyping, even if the method is not called.

Both of these problems are also applicable for conditional signature expansions where both problems can occur at runtime if the signature of the type has not been expanded and thus an invalid call is made.

7.4 Locations for extra checking

Before discussing the recommendations that should solve the problems introduced by signature modification, we will first look at the locations where possible solutions can be implemented. These locations are the Compose⋆ tool itself, around the .NET compiler and at runtime during program execution. We will give the advantages and disadvantages for each of these locations

Compose⋆ at compile-time

This is the preferred location for any additional checking. There are two reasons for this: first, it is the earliest possible place in the compile process. Second, we have the most control over this part, since we can modify Compose⋆. The tool can be extended to include checking for any new errors we have introduced.

A possible downside is that implementing all new error checking into Compose⋆ might require more work than other solutions, since we must implement functionality that could be available in other parts. An example of this is type checking of the source code which would require writing our own compiler.

Another problem is that it can be impractical or even impossible to find all possible errors at this time in the process. An example of this is conditional signature modification, which depends on conditions that can only be evaluated at runtime.

Around the .NET compilers

The big advantage of using the standard .NET compilers is that we can let the compiler do all the work instead of having to implement the functionality ourselves. By controlling the input and output of the compiler, we can use the compiler to do all the work.

An example of this is catching error messages that we know to have been introduced by composition filters. If we recognize one of these error messages, we can extend it with information regarding the composition filter that has introduced this error.

Wrapping our own code around the .NET compiler does mean we are dependent on the behavior of the .NET compilers. Since the different compilers possibly all return different output, it requires implementing individual wrappers for each compiler. Furthermore, changes to the compilers by Microsoft might break our implementations.

Like with modifying Compose*/.NET itself, another problem is that it can be impractical or even impossible to find all possible errors at this time in the process. The example of conditional signature modification applies here too.

Checking at runtime

Any checking at runtime will be a necessity rather than an implementation choice. We want to avoid problems occurring at runtime as much as possible. Problems occurring at runtime are more easily missed during development and might confront end-users of the program after its release.

Unfortunately, it might be unavoidable for some problems. Already mentioned is the conditional signature modification example. The modification of the signature is done at runtime and depends on values that are only known at runtime. This means that calls to methods can fail at runtime and these calls must be handled properly.

7.5 Recommendations

The recommendations come in two pairs; two recommendations to solve the problems at compile-time and two to solve the problems at runtime.

Compile-time

There are two extensions necessary to the compilation process to solve the problems at compile-time. First, to avoid the unclear error and warning messages returned by the standard .NET compilers, we must extend these messages to include information about the filter that introduces this behavior.

This can be achieved by intercepting all warnings and errors returned by the compiler. If the warning or error is one of the messages that could be the result of code introduced by a filter, then we check if the message indeed mentions a method that was added or removed by a filter. If this is the case, we extend the message with the needed filter information before displaying it to the developer. Modifying messages from the standard

compiler is easier than extending Compose⋆/.NET to incorporate these checks, but this approach is compiler dependent.

The second fix at compile-time is that Compose⋆/.NET needs to be able to recognize conditional signature modifications. Should a conditional modification be encountered, then a warning has to be displayed. The warning has to inform the developer where the conditional modification is applied and what the runtime effects might be. This warning should always be displayed, since checking the validity of calls to methods that are affected by conditional modifications is impossible at compile-time.

Runtime

At runtime, we need to introduce functionality to handle calls to methods that have been removed or have not been added by signature modification. There are two methods for handling these invalid calls: fail silently or throw an exception.

Failing silently means that a called method simply returns if the call to the method is invalid. If a method requires a return value, then the default value will be returned. The advantage of this approach is that the program can continue when an invalid call is made.

The big disadvantage is that the invalid call actually goes unnoticed. The invalid call can introduce errors and unexpected behavior and since the calling code cannot recognize the failing call, these errors and unexpected behavior can migrate and manifest themselves at other locations in the program. This migration complicates debugging and thus development.

The other approach is to throw an exception when an invalid call is made. If this exception is not caught, then the program will exit with an error message. The advantage of this approach is that the error cannot be ignored by the developer and the error message can help in locating and correcting the problem. By throwing an exception that can be caught, developers can write code that actually takes invalid calls into account and can continue executing correctly even if an invalid call has been made.

The fact that failing silently can introduce unexpected behavior (in worst cases long after the invalid call) and makes finding and correcting the error unnecessarily difficult means that throwing an exception is the preferred approach for handling invalid calls.

The second problem at runtime is recognizing when conditional signature modification breaks the program. Examples of this are the last two problems described in the previous section. Another example of this problem is an invalid call to a method that is inherited and implemented in a subtype. If the call to the method is invalid, should we throw an exception or call the method in the super type.

The second problem is solved by a new approach to signature reduction. Presently, signature reduction involves removing a method from the type's signature by removing the method from the type. This approach is now changed to removing a method from the signature of a type by making the calls to the method invalid at runtime.

This solves all problems at runtime with conditional signature reduction (and expansion). Since removing the method from the type's signature now means that only calls to the method are made invalid, the only check that needs to be performed is the validity

of the call to this method. Whether the method is part of an inheritance structure or is called through inheritance is irrelevant.

This new approach to signature reduction does introduce a new problem; the current implementation of unconditional signature reduction does not follow the new approach. The current implementation removes the method completely from the type at compile-time, thus making all calls to the method invalid. This does not follow the new approach to signature reduction, which makes calls invalid, but leaves the method in place.

Therefore, unconditional signature reduction has to be changed. Instead of removing the method at compile-time, the code of the method has to be changed to always throw the new invalid call exception. This means that calls to this method can no longer be checked at compile-time, but will instead throw the same runtime exception as conditional signature modification. The same warning, as is given with conditional signature modification, should be displayed at compile-time to alert the developer.

Chapter 8

Behavioral subtyping

In the previous chapters, we have focused on signature modification. Signature modification is applied to modify the signature of a type. Besides having a signature, a types also has a behavioral signtature: this is all the observable behavior exhibited by an object of the type. Composition filters not only modify signatures, but can also modify the behavior of types and methods. This is called behavioral signature modification.

The type containing the modified method can be seen as a subtype of the original type. These subtypes could potentially introduce unexpected behavior to the original types. To prevent this, a form of type checking can be applied. The checking of these subtypes against the original typesis called behavioral type checking. By checking the behavior of the subtype againts the behavior of the original type, we can recognize (potential) behavioral conflicts.

In this chapter, we discuss behavioral subtypes and type checking. First, we give an introduction into behavioral subtypes. This includes a formal definition for the behavioral subtype relation and two exapmples of approaches to describing the behavior of a program. Then, we introduce the model that is already in use in Compose⋆ to describe behavior: the *resource-operation* model. Third, we look at a number of approaches to determine the behavior of a program. Finally, we evaluate the possibilitiy of using the resource-operation model to represent and reason about the full behavior of a program.

8.1 Introduction behavioral subtypes

Before we look at behavioral subtypes and type checking, we will first look at normal subtypes and type checking. A definition for a subtype can be stated as follows:

> A type is a valid subtype if the signature of the type contains the signature of the supertype. The signature of a type is the set of visible methods that define the possible interactions with an object of the type.

The fact that a valid subtype always contains the signature of its supertype means that any method defined in the supertype is defined in the subtype. This allows for the

interchanging of an object of the supertype with an object of the subtype without calls to methods resulting in runtime errors.

If we look at the code in Listing 8.8, then we can see an example of a valid subtype. `SubClass` inherits from `SuperClass` and is therefore a subtype of `SuperClass`. The `SuperClass` type only contains the `Test` method and this method is also defined in `SubClass`. This means that both types have the same signature and `SubClass` is therefore a valid subtype of `SuperClass`.

```
namespace Example {
  class SuperClass {
    protected List<int> _items = new List<int>();

    public virtual void Add(int i) {
      _items.Add(i);
    }

    public int Get() {
      return _items[0];
    }
  }

  class SubClass : SuperClass {
    public override void Add(int i) {
      _items.Insert(0, i);
    }
  }

  class Test {
    public static void Main() {
      SubClass sub = new SubClass();
      SuperClass super = sub;
      super.Add(0);
    }
  }
}
```

Listing 8.1: Example of subtyping

However, if we look at the behavior of the two `Add` methods, then we can see that the two methods behave in a different manner. The `Add` method in `SuperClass` adds the integer to the end of the list whereas the `Add` method in `SubClass` adds the integer to the top of the list. This gives the `SuperClass` the behavior of a queue, while `SubClass` has the behavior of a stack.

So, if we look at this subtyping from a behavioral point of view, then we can see that both methods (and therefore both types) potentially have different behaviors. `SubClass` does not appear to be a valid behavioral subtype of `SuperClass`. This indicates that a different definition is needed for behavioral subtypes.

Therefore, a formal definition for the behavioral subtype relation is given next. To apply the formal definition of a behavioral subtype, we need to describe our implementation in terms of its behavior. After the formal description, we describe two examples

of existing behavioral description formats (*Design by Contract* and *Larch/C++*).

8.1.1 Definition for the behavioral subtype relation

Generally speaking, a type is valid behavioral subtype if it does not replace any of the behavior exhibited by the supertype and it does not introduce behavior that contradicts the behavior exhibited by the supertype. An object of a behavioral subtype can replace an object of its behavioral supertype without any unexpected behavior being observed when calls are made to methods defined in the supertype.

A formal definition of the behavioral subtype relation can be found in [31].

Definition of the subtype relation , $\preceq$: $\sigma = \langle O_\sigma, S, M \rangle$ is a subtype of of $\tau = \langle O_\tau, T, N \rangle$ if there exists an abstraction function, $A : S \rightarrow T$, and a renaming map, $R : M \rightarrow N$, such that:

1. Subtype methods preserve the supertype methods' behavior. If m_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:

   - *Signature rule.*
     - *Contravariance of arguments.* m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α_i and that of m_σ is β_i, then $\forall i. \alpha_i \preceq \beta_i$.
     - *Covariance of result.* Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be α and m_σ's be β. Then $\beta \preceq \alpha$.
     - *Exception rule.* The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.
   - *Methods rules.* For all $x : \tau$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow \mathrm{m}_\sigma.pre$.
     - *Pre-condition rule.* $m_\sigma.post \Rightarrow \mathrm{m}_\tau.post[A(x_{pre})/x_{pre},\ A(x_{post})/x_{post}]$

2. Subtypes preserve supertype properties. For all computations c, and all states ρ and ψ in c such that ρ precedes ψ, for all $x : \sigma$:

   - *Invariant rule.* Subtype invariants ensure supertype invariants.
     $I_\sigma \Rightarrow \mathrm{I}_\tau[A(x_\rho)/x_\rho]$
   - *Constraint rule.* Subtype constraints ensure supertype constraints.
     $C_\sigma \Rightarrow \mathrm{C}_\tau[A(x_\rho)/x_\rho,\ A(x_\psi)/x_\psi]$

A type is defined as a triple: a set of objects (o other types), a set of values (objects of primitive types) and a set of methods. The abstraction function (A) is used to link a method in the subtype with its corresponding method in the supertype, allowing both methods to be compared. The renaming map (R) is used to connect a value in the

subtype with its corresponding value in the supertype, thus allowing the two values to be compared.

There are three signature rules to guarantee that a subtype method preserves the behavior of the equivalent supertype method. First, both methods must have the same number of arguments and each argument of the subtype method must be a behavioral subtype of the argument in supertype method. Second, the result type of the method in supertype must be a valid behavioral subtype of the result type of the method in the subtype. Third, all possible exceptions thrown by the subtype method must be part of the set of all possible exceptions thrown by the supertype method.

It is possible that the first two signature rules are enforced by the programming language used for the implementation. For example, both C# and J# require overriding methods in a subtype to have the same argument types and the return type as the method in the supertype. If the subtype method has different arguments, it overloads the supertype method and both methods will exist in the subtype. Methods with the same method name and arguments but different return types are not allowed at all.

There are two methods rules to guarantee that a subtype method preserves the behavior of the equivalent supertype method. First, each precondition of the subtype method must be present in the supertype method; a subtype method cannot introduce new preconditions. Second, each postcondition in the supertype method must be present in the subtype method; a subtype method cannot ignore any postcondition in the supertype method.

There are two rules that ensure that a behavioral subtype preserves its supertype properties. First, the invariant rule states that any invariant in the supertype is upheld by the subtype. Second, the constraint rule states that the subtype cannot break any constraints put in place by the supertype. These rules prohibit any new methods in the subtype from having functionality that introduces unexpected behavior for members defined in the supertype.

Two modifications were introduced [8]: a redefinition of the exception rule and a redefinition of the contstaint rule.

Definition of the exception rule:

- *Exception rule.* The sets of exceptions signaled by m_σ and m_τ are respectively E_σ and E_τ. For each e_σ in E_σ, there is an e_τ in E_τ and $e_\sigma \preceq e_\tau$.

The new definition of the exception rule describes the first modification. This is a modification to the exception rule allows as subtype method to throw exceptions that are a behavioral subtype of one of the exceptions thrown by supertype method. This allows for exception inheritance as is supported by many programming langauges, such as C#.

Definition of the constraint rule:

- *Constraint rule.* For all valid computations c, which do not invoke extra methods of σ, and all states ρ and ψ in c such that ρ precedes ψ, for all $x : \sigma$:

$$C_\sigma \Rightarrow \mathrm{C}_\tau[A(x_\rho)/x_\rho,\ A(x_\psi)/x_\psi]$$

The new definition of the constraint rule describes the second modification; a modification of the constraint rule. This modified constraint rule allows methods added to the subtype to ignore the constraints imposed by the supertype. The idea being that these methods are hidden when the subtype is accessed through an object of the supertype and the constraints cannot be violated through the supertype object.

The modified constraint rule does require some new restrictions. For example, manipulating an object simultaneously from the subtype's and the supertype's point of view could result in unexpected behavior. This can occur when simultaneously referencing the same object through an object of the supertype and through an object of the subtype.

This modification of the constraint rule weakens the rule and the definition of the behavioral subtype relation. Therefore, the definition with the modified constraint rule is called *weak behavioral subtyping*. The definition with the original constraint rule is called *strong behavioral subtyping*.

It is important to note that this definition applies strictly to behavior and not implementation. Behavior follows from implementation, but multiple different implementations can exhibit the same behavior. Listing 8.2 shows two implementations of an add function. They are two different implementations, but they both exhibit the same behavior; the parameters are added up and the result is returned.

```
1 unsigned int Add(unsigned int x, unsigned int y) {
2   return x + y;
3 }
4
5 unsigned int Add(unsigned int x, unsigned int y) {
6   for (int i = 0; i < y; i++)
7     x++;
8   return x;
9 }
```

Listing 8.2: Example of different implementations with the same behavior

8.1.2 Design by Contract

An approach that can describe behavior is *Design by Contract* [12]. This approach defines behavior by establishing a contract between the caller and the callee. This contract defines the responsibilities of both parties involved.

A typical contract between the caller and the called method consists of three components:

Preconditions: these are conditions that the called method expects to be fulfilled at the time the method is called.

Postconditions: these are conditions that the called method guarantees to be fulfilled after it has executed.

Class invariants: these are constraints on members (such as fields) of a class that have to be maintained by all methods of the class.

There are a number of rules that apply to inheritance to ensure that overriding methods in subtypes do not break contracts defined in their supertypes:

- Preconditions may be weakened in subtypes but not strengthened.
- Postcondition may be strengthened in subtypes but not weakened.
- Class invariants may be strengthened in subtypes but not weakened.

These rules ensure that subtypes can only create stricter contracts than the contracts defined in their supertypes. A subtype can therefore not introduce behavior that is prohibited by the contract in its supertype.

An example of a programming language that depends on the Design by Contract approach is *Eiffel* [11]. The Design by Contract approach is implemented in the Eiffel programming language through a number of language constructions.

```
1  class ACCOUNT create
2    make
3  feature
4    minimum_balance: INTEGER -- Minimum permitted value for balance
5    deposit (sum: INTEGER) -- Deposit sum into the account.
6      require
7        positive: sum > 0
8      do
9        add (sum)
10     ensure
11       deposited: balance = old balance + sum
12     end
13 invariant
14   sufficient_balance: balance >= minimum_balance
15 end
```

Listing 8.3: Example of Design by Contract in Eiffel

Listing 8.3 describes a part of an implementation of a bank account in Eiffel. The `ACCOUNT` class contains a `deposit` procedure, which is responsible for depositing a sum of money on the account. The procedure has one precondition indicated by the `requires` keyword and one postcondition indicated by the `ensures` keyword. Finally, the class contains one class invariant indicated by the `invariant` keyword.

The contract that this method has with the caller consists of three parts. First, the method expects the sum deposited to be larger than zero. In return, the method guarantees that the balance after the deposit is the old balance with the deposited sum added. Finally, the class guarantees that the balance will always be above a certain minimum.

By using Design by Contract, Eiffel provides the functionality to describe the behavior of types and methods and a way to enforce this behavior onto subtypes. However,

Eiffel does not comply with the behavioral subtyping relation. It violates the contravariance of arguments rule by allowing arguments of methods to be covariant; arguments of an overriding method in a subtype can be a subtype of the original argument. This means that Eiffel cannot fully enforce behavioral subtypes. Eiffel is still a good example of a programming language that contains support for describing and enforcing behavior.

8.1.3 Larch/C++

Larch [29] is not a programming language, but rather an approach to extend an existing programming language with functionality for formally reasoning about program components that are implemented in the programming language. The Larch family of languages supports a two-tiered, definitional style of specification[17]. A Larch specification comes in two parts (or tiers): the *BISL* and the *LSL*.

The first part (or top tier) of any Larch specification is the *behavioral interface specification language* (BISL) or *Larch interface language*. This part of the specification is tailored specifically to the supported programming language and models the interface and behavior of the program modules. The BISL typically uses the programming language itself to define interfaces and added annotations to define behavior.

The second part (or bottom tier) of a Larch specification is the *Larch shared language* (LSL). The LSL is used to specify a mathematical vocabulary with models and auxiliary functions. These models and functions are written in manner that is independent of any programming language. The use of independent mathematical operations makes the formal reasoning about the semantics of program modules more concise. In short, the LSL provides the mathematical vocabulary with which the behavior of the program modules is defined in the BISL.

Because Larch is an approach to extending existing programming languages, a family of Larch implementations has evolved. One of the members of the Larch family is *Larch/C++* [30]. Larch/C++ extends the C++ programming language with the described Larch functionality. The reason for Larch/C++ as an example is that of all Larch-extended programming languages, C++ is the most similar to the Compose⋆/.NET supported languages in terms of syntax and concepts. We first describe an example of a Larch/C++ BISL, before describing an example of a very simple LSL.

```
1  //@ uses IntHeapTrait;                        // connection to LSL
2
3  /*@ abstract @*/ class IntHeap {             // no constructors, C++
       interface
4  public:                                      //
5    virtual int largest() const throw() = 0;// C++ interface
6    //@ behavior {                             // starts largest's
         specification
7    //@   requires len(self^) >= 1;           // precondition
8    //@   modifies nothing;                    // what can change
9    //@   ensures result = head(self^);       // postconditon
10   //@ }
```

```
11 };
```

Listing 8.4: An example of a Larch/C++ BISL

Listing 8.4 shows a Larch/C++ BISL which describes the interface of an abstract `IntHeap` class with only one method: `largest`. The interface of the class is specified in C++, while the annotations describing the behavior are added as comments. This allows the BISL to be used as standard C++ source code, in which case the behavior specification is simply ignored.

The behavior of the `largest` is defined by a precondition, an invariant and a postcondition. The precondition (`requires`) states that the heap must contains at least one element. The invariant (`modifies`) states that nothing may be modified when executing the method. The postcondition (`ensures`) guarantees that the returned integer is the fist element on the heap.

```
1 % This is LSL, not Larch/C++
2 IntHeapTrait: trait
3   includes PriorityQueue (int for E, IntHeap for C, int for Int),
4            NoContainedObjects(IntHeap)
```

Listing 8.5: An example of a Larch LSL

Listing 8.5 shows the specification of the `IntHeapTrait` trait. The *trait* is the basic unit of specification in LSL. A trait can introduce operators and some of their properties. `IntHeapTrait` includes two other traits (`PriorityQueue` and `NoContainedObjects`). By including these two traits, `IntHeapTrait` combines the vocabularies of the two traits to define the vocabulary that can be used to define the behavior of the `IntHeap` class in the BISL. For example, `len` and `head` are defined in `PriorityQueue` or one of the traits included by `PriorityQueue`.

The `PriorityQueue` trait that is included by the `IntHeapTrait` describes the behavior of a queue which has an ordering on its elements. `PriorityQueue` is defined with abstract data types and a translation is therefore required, translating the elements of the queue to C++ integers, the container to the IntHeap implementation and the abstract integer data type to the C++ integer. The `NoContainedObjects` trait is included to describe that the abstract values of the `IntHeap` class do not contain any objects, only values.

The Larch/C++ approach is an example of an approach that allows for the description of behavior and enforcing behavioral subtyping. With support for defining invariances and constraints for types, both the strong and the weak behavioral subtype relations can be applied. Larch/C++ is also an example of extending an existing programming language with behavioral describtion functionality, while keeping the implementation in the source code backwards compatible.

8.2 The resource-operation model

Current behavioral analysis in Compose⋆ describes behavior in a different description format: the *resource-operation model*. The resource-operation model allows for the representation of both concrete and abstract behavior in terms of resources and operations

performed on those resources. Behavior can be represented as a sequence of resource-operation pairs. By comparing these sequences to any defined constraints, we can reason whether the behavior is allowed or not. The resource-operation model consists of three elements:

Resources A resource represents an element on which operations are performed. Resources can represent concrete elements such as variables in source code, but can also represent more abstract elements. The definition of a resource consists of two parts:

- A name identifying the resource.
- A vocabulary that defines the operations allowed on the resource.

Operations An operation is defined for and performed on a resource. Like resources, an operation can represent a concrete action (for example, a method call) or it can be more abstract. Every operation has a name as an indentifier. A possible extension of the model could be to allow operations to have parameters, but these are not used in the resource-operation model used in Compose⋆.

Constraints A constraint is a pattern that enforces a sequence of allowed or disallowed operations. Constraints can be in the form of conflict rules or assertion rules. A conflict rule is a pattern that defines a combination of disallowed operations. An assertion rule is a pattern that enforces a sequence of operations. There are three parts to a constraint:

- A set of resources that is targeted by this constraint.
- A sequence of resource-operation pairs that describes the constraint. Regular expressions can be used in the constraint sequence.
- A message can be defined that explains the constraint.

As an example of the resource-operation model, we give the model that is used by *SECRET* (SEmantiC Reasoning Tool). SECRET is part of Compose⋆ and performs an analysis of the applied filters to see if any of the composition filters possibly conflict due to the behavior of the filters. it is described in more detail in Section 8.3.1.

```
1  Resource: arg
2  Vocabulary: discard read write
3
4  Resource: message
5  Vocabulary: discard dispatch exit return
6
7  Resource: return
8  Vocabulary: discard read write
9
10 Resource: selector
11 Vocabulary: read write
12
```

```
13 Resource: target
14 Vocabulary: read write
```

Listing 8.6: Resources used by SECRET

Listing 8.6 shows the resources used by SECRET. The important resources are `arg`, `message` and `return`. The `message` resource represents the intercepted message. The `arg` resource represents the arguments of the message. The `return` is the return value of the message.

```
1  Rule # 1
2  Rule type: constraint
3  Resource: message
4  Pattern: (exit)(return|dispatch)
5  Message: Can not return or dispatch a message after an exit.
6
7  Rule # 2
8  Rule type: constraint
9  Resource: *
10 Pattern: (discard)![exit]+
11 Message: Discarded resources can not be accessed.
12
13 Rule # 3
14 Rule type: constraint
15 Resource: message
16 Pattern: (return)(return)
17 Message: A message can be returned only once.
```

Listing 8.7: Constraints used by SECRET

The constraints are described in Listing 8.7 as rules. There are only three rules and they are all fairly straight forward. All three rules are conflict rules (which in SECRET are called constraints), which means that the sequences defined by the rules cannot be present in any sequences that define behavior.

```
1 arg.read, message.dispatch, return.read, message.return
2 arg.read, message.discard, return.read, message.return
```

There are two sequences of resource-operation described above. The first sequence describes a behavior that is allowed, since it does not conflict with one of the constraints. The second does conflict with a constraint; the `message.return` after the `message.discard` is not allowed by the second constraint.

8.3 Determining the behavior

Before we can reason about the behavior of a program, we need to determine the exact behavior of the program. We discuss three approaches to determining this behavior:

- Through filter analysis; a process already implemented in Compose⋆ that analyzes the behavior of the applied filters.

- Through source code analysis; an automated process that parses and analyzes the source code to extract the behavior of the program.

- Let the developer define the behavior of the program; a number of language constructions are added to Compose⋆ to allow the developer to define the behavior of the source code. No source code analysis is required with this approach.

The first approach (filter analysis) relies on the behavior of the applied filters alone to determine the behavior of the program and to find any possible behavioral subtypes and behavioral conflicts. The second approach (source code analysis) uses the source code to determine the behavior and does not require any extra information. The third approach relies on the developer to define the behavior. This raises the question how and where this behavior is to be defined. After looking at the souce code analysis process, we look at a number of extensions that provide the developer with the ability to define the behavior. These extensions are behavior descriptions separate from the source code and behavior descriptions using custom attributes.

One behavior defining approach will not be discussed here: extending the programming languagues with special language constructions. Extending a programming language would mean that Compose⋆ can no longer be applied transparently. Furthermore, extending any of the programming languages is not supported by the used .NET compilers and custom language construction would make the source code no longer re-usable in other projects.

The first approach gives a behavioral description of the filters. Because this approach is already implemented, the format of the description will be the resource-operation model. The two other approaches result in a behavioral description of the types and their methods in the source code. The format of this description is not set, but any examples will be described with the one of the formats explained in Section 8.1 and Section 8.2.

8.3.1 Filter analysis

We look at the only behavioral analysis currently present in Compose⋆: filter analysis. The tool that analyzes the filters in Compose⋆ is *SECRET* (SEmantiC Reasoning Tool). SECRET performs an analysis of the applied filters to see if any of the composition filters possibly conflict due to the behavior of the filters.

Conflicts can occur between filtermodules that are imposed on the same jointpoint. Because the concerns that define these filtermodules may have been developed independently, they could introduce side effects that only affect other filtermodules. If these concerns are combined on the same joint points, semantic conflicts could become apparent. SECRET performs a static analysis on the semantics of the applied filters and detects possible conflicts.

We look at two examples and the results of the analysis by SECRET to determine if filter analysis alone can be used to reason about possible behavioral subtypes. The two examples that are described are subtyping through dispatch filters and a method

extension through a before of after filter. Finally, we evaluate the examples to see if SECRET is able to reason about behavioral subtypes.

Subtyping introduced through dispatch filters

The first example we describe, is the creation of subtypes through dispatch filters. Subtyping through dispatch filters involves a composition filter that expands the signature of the intended subtype with the methods of the intended supertype. The modified type now contains the signature of the supertype making it a subtype.

```
namespace Test {
  class B
  {
  }

  class A
  {
    public int Test()
    {
      return 0;
    }
  }
}
```

Listing 8.8: Two unrelated classes

```
concern SubtypeTesting in Test
{
  filtermodule SubtypeTest
  {
    internals
      a: Test.A;
    inputfilters
      add: Dispatch = { [*.Test] a.Test }
  }

  superimposition
  {
    selectors
      subjects = { C | isClassWithName(C, Test.B') };
    filtermodules
      subjects <- SubtypeTest;
  }
}
```

Listing 8.9: Dispatch filter introducing B as a subtype of A

Listing 8.8 shows two classes, `A` and `B`, with no relation between them. We use a composition filter to make `B` a subtype of `A` which is shown in Listing 8.9. The filter defines a dispatch filter on `B` which filters on messages for a `Test` method. Any messages for this will be dispatched to `a.Test`. The result is that any calls for `B::Test()` are dispatched to `A::Test()`. This makes `B` a subtype of `A`, since they share the same signature.

Output by SECRET

From the output of SECRET, we can create two sequences of resource-operation pairs. We can ignore the second sequence, because this is the sequence for the default action (the action performed if no filter accepts the message). This action simply dispatches the message to the targeted method.

```
Test selector: arg.read, message.dispatch, return.read, message.return
  + selector: arg.read, message.dispatch, return.read, message.return
```

The first sequence applies to the `Test` method (as indicated by the Test selector). This sequence describes the behavior of a dispatch filter. It shows that the arguments are read, the message is dispatched (but not where to), the return value is read and the message is returned.

For class `B` to be a subclass of `A`, all methods of `A` should be dispatched to by the composition filters applied to `B`. All dispatches on `B` should be analyzed for their substitute target and selector; these have to be compared to the types and methods in the source code. If there are dispatches to all methods in class `A`, then it could be considered a superclass of `B`. The resource-operation sequences described above are not needed for this analysis.

For `B` to be a behavioral subtype of `A`, the previous rule should apply (`B` has to be a subtype of `A`) and `B` cannot contain any methods that somehow conflict with the observable behavior of class `A`. This last requirement can only be checked if a representation of the behavior of both `A` and `B` is present. Because SECRET only looks at the behavior of filters, this information is not available.

Subtyping introduced through method extending

A (behavioral) subtype of a type can be introduced by a composition filter by adding a `Before` or `After` filter. By applying one of these filter types to a method, we can extend the functionality of the method. The type with the composition filter applied could be considered a subtype of the original unmodified type.

```
namespace Test {
  class A
  {
    public void Test(int i)
    {
    }
  }

  public class C
  {
    public void Log (JoinPointContext jpc)
    {
      for (short s = 0; s < jpc.ArgumentCount; s++)
        Console.Out.WriteLine(jpc.GetArgumentType(s).ToString());
    }
  }
```

```
}
```

Listing 8.10: Example of a logging class

```
concern MethodTesting in Test
{
  filtermodule ExtendedMethod
  {
    externals
      c: Test.C;
    inputfilters
      log: Before = { [*.Test] c.Log }
  }

  superimposition
  {
    selectors
      subjects = { C | isClassWithName(C, 'Test.A') };
    filtermodules
      subjects <- ExtendedMethod;
  }
}
```

Listing 8.11: Before filter extending Test method in class A

Listing 8.14 shows class `A` with the `Test` method which will be extended with the `Log` method in `C`. The method is extended by the concern described in Listing 8.15, which defines `Before` filter on the `Test` method. This filter ensures that the `Log` method in `C` is called before `Test`. This form of extending can also be achieved by using an `After` filter.

Class `A` with the extended `Test` method could be seen as a subclass of `A` without the extended method, with both sharing the same type signature. The extended class `A` could also be seen as a behavioral subtype of the original class `A`, since the extra logging does not introduce any unexpected observable behavior.

`Before` and `After` filters can modify the passed arguments and return value. The filters could cause runtime errors by removing arguments passed to the method or by changing the type of the return value returned to the calling code.

`Before` and `After` filters can also introduce unexpected behavior to a method. For example, an `After` filter could modify the return value, making it break a behavioral post-condition of the orginal method. This means that the subtype introduced by the composition filters is not a valid behavioral subtype of the original type.

To recognize both problems, it is not enough to know the `Before` or `After` filter is present and dispatches to another method. We need a representation of the behavior of the method called by the `Before` or `After` filter and a representation of the behavior of the original method.

Output by SECRET

Like the output from the previous example, we can create two sequences of resource-operation pairs. We can again ignore the second sequence, because this is the sequence

for the default action (the action performed if no filter accepts the message). This action simply dispatches the message to the targeted method.

```
1 Test selector: arg.read, message.dispatch, return.read, message.return
2   + selector: arg.read, message.dispatch, return.read, message.return
```

The first sequence applies to the `Test` method (as indicated by the Test selector). This sequence describes the behavior of a `Before` filter. It shows that the arguments are read, the message is dispatched (but not where to), the return value is read and the message is returned. This sequence could also represent the behavior of an `After` filter.

If we look at the sequence, we notice two things: first, the sequence is the same as the sequence for the `Dispatch` filter. Second, there is no way to reason about the effect of the called method by the filter (the `Log` method) on the targeted method (`Test`).

The fact that we cannot distinguish between a `Before` and `Dispatch` filter is because the two filters perform the same action. They both dispatch the message to another method. The difference between the two filters is that a `Dispatch` filter will return, whereas a `Before` will call the original targeted method. However, this difference cannot be deduced from the resource-operation sequences.

We cannot reason about the effect of the called method by the filter, because we do not know the behavior of the called method or the behavior of the original method. Without either behavior, it is impossible to reason about any possible semantic conflicts between the two methods. Therefore, it is impossible to determine whether the subtype is a valid behavior subtype of the original type.

Evaluation of the behavioral analysis by SECRET

With the information provided by SECRET, only the check if a type is a subtype through dispatching could be implemented in SECRET. All other checks require more information about the behavior of the targeted and added methods. Without this information, it is impossible for SECRET to reason about the behavior of the subtypes by the composition filters.

A number of steps have to be introduced for this information to become available. First, the behavior of the involved methods has to be known. Second, this information would have to be combined with the behavioral representation of the applied filter to create the behavior of the new subtype. The behavior of the new subtype can be compared against the behavior of the original method.

8.3.2 Source code analysis

Source code analysis (or *static code analysis*) is an approach that attempts to deduce the behavior of a program by analyzing its source code. The term static indicates that the program will not evaluated by executing it.

There are two advantages to the static analysis of the source code. First, the analysis of the source code can be done automatically. This takes the responsibility away from the developer and would guarantee a true and consistent representation of the behavior of the program.

The second advantage is that it allows the use of any programming language to write to the source code, including existing programming languages. Because the source code itself is analyzed, there is no need for any extensions to the languages or other constructions to define the behavior.

Unfortunately, there is one big disadvantage to using static code analysis. Static code analysis has been proven to be undecidable for finding all possible behavior of a program. There is no mechanical method that can always answer whether a given program will exhibit runtime errors.

This proof was found in the 1930s and an example of the problem was given by Turing in [54]. This problem is called *the halting problem*. The halting problems states the following: given a program and a finite input, decide whether the program will finish or will run infinitely, given that input. Turing proved that a general algorithm that could prove this for all possible program-input pairs cannot exist.

This undecidability means that a tool to analyze a program by looking at the source code, can never garantuee to fully predict the behavior of the program for all possible input. The found behavior of a program can therefore never be garantueed to be correct. This behavior would not be suitable for reasoning with.

Altough an approximate behavior of the program could be found through a static code analyzing tool, but this behavior could be too general or the analysis of the behavior could potentially be a very time-consuming task.

A second disadvantage is determining whether the behavior describing the source code is the intended behavior. The found behavior could be the result of a programming error rather than the intended behavior. This cannot be determined by the source code analysis itself.

8.3.3 Separate behavior specifications

The first approach that relies on the developer to define the behavior of the implementing source code uses specifications that are written completely separate from the source code. A custom behavior definition can be used for these descriptions. As an example for separate behavior specifications, we will look at Larch which was described in Section 8.1.3.

There are three locations for the behavior definitions:

- In the source code. The descriptions are added to the source files. To allow the source code to compile, the descriptions must either be added as comments or have to be removed from the source files before the files are compiled. The BISL of Larch/C++ is an example of this approach.

- In a separate behavior description file. The descriptions are located in a separate file, which is used to analyze the behavior of the implementation. The LSL definitions of Larch use this approach.

- A combination of the two previous locations. Behavior definitions can be found in the source files and in separate description files. Larch/C++ uses this approach by

putting the BISL definitions in the source code and the LSL definitions in separate files.

Writing the behavior definitions in the souce code has the advantage that the source code itself can be used to help define the behavior. For example, Larch/C++ uses the C++ method definitions to enforce the Signature rules. A disadvantage of this approach is that the code is less readable, especially for developers that do not use the behavior definitions.

Using separate behavioral definition files allows the source code to remain unmodified. The separate files can also be re-used and can accompany compiled source code (for example, to describe the behavior of a library). The disadvantage of this approach is that the behavioral defintions must include the signature of methods, which may also be defined in the source code.

The combination of the two approaches attempts to capture the advantages of both approaches. By writing parts of the behavior description in the source files, we can use the source code to define part of the behavior. By also using separate behavior description files, we can reuse behavioral descriptions. Combining the two approaches does suffer from the disadvantage of the first approach; the source code is less readable, due to the behavioral descriptions added.

8.3.4 Custom attributes

The .NET Framework provides functionality to write custom attributes to describe additional properties of a type or method. A number of custom attributes could be conceived that can be added to types and method to describe their behavior. Examples of custom attributes that describe behavior can be found in [50].

```
1
2 [Semantics("song.credit")]
3 public void withdraw ( ReifiedMessage message )
4 {
5   // song.credit
6   this.numCredits--;
7   Console.WriteLine("Payed, " + numCredits + " left");
8   message.resume();
9 }
```

Listing 8.12: Example of custom attributes describing behavior

Listing 8.12 shows an example of a custom attribute that describes the behavior of a method with the resource-operation model. The method `withdraw` has one custom attribute that describes the semantics of the method. It states the `credit` operation is performed on the `song` resource. This performed in the method by the line `this.numCredits--;`. The other functionality of the method does not exhibit any observable behavior and are therefore not describes by the `Semantics` attribute.

By using custom attributes to describe behavior, we can leave part of the checking of the validity of the descriptions to the .NET compilers. If the compilers can compile

the code with the custom attributes, then we know that the descriptions are written in a valid syntax.

There are a number of disadvantages to using custom attributes. The custom attributes have to be added to the source code, although a combination of custom attributes and separate specification files is possible. The general form of custom attributes is set; a name with a number of parameters. Custom attributes may not be properly supported in every .NET language supported by Compose⋆/.NET. Finally, the custom attributes must be defined in a library. This library must be referenced by the project using the source code with the attributes, making the source code less reusable.

8.4 Evaluation of behavioral analysis using the resource-operation model

We have seen that composition filters can introduce (behavioral) subtypes. The (behavioral) subtypes are created from the original types by applying the composition filters. By comparing the behavior of the original type and the new subtype, we can determine whether the new subtype exhibits conflicting behavior in comparison with the orginal type and thus whether the composition filter introduces unexpected behavior.

Current behavioral analysis in Compose⋆ is limited to the filter analysis by SECRET (Section 8.3.1). Unfortunately, we have shown that filter analysis alone is not enough to detect a composition filter introducing conflicting behavior.

To detect conflicting behavior being introduced by the composition filters, we need to include the behavior of the source code that is introduced by the composition filter, as well as the behavior describing the original source code.

This requires the current behavioral representation to be extended to include the describing of behavior of the program. The current behavioral representation uses the resource-operation model. To simplify the implementation of an extended behavioral representation and the combination with filter analysis, we will use the resource-operation model for a full description of the behavior of the program.

We look at two possible approaches to the detection of behavioral conflicts. First, we look at the feasability of implementing full behavioral analysis of a program including any composition filters that are applied. Second, we describe an alternative to full behavior analysis. This alternative does not support full behavioral analysis of a program, but should allow the detection of possible behavioral conflicts by a composition filter.

8.4.1 Full behavioral analysis

The ultimate goal would be the description of the behavior of the complete program, including any applied composition filters. With this description, we then want to reason about the behavior of the program and the existance of any behavioral subtypes. A full behavioral analysis should make it possible to decide whether any behavioral conflicts will occur.

Since we are looking at extending the existing behavioral analysis in Compose★, we want to use the resource-operation model as the behavioral description format for the analysis. This leads to two questions:

- Can we describe all behavior with the resource-operation model.
- Can we reason about behavioral subtypes based on this behavioral description.

To answer these questions, we look at an example of source code behavior being described by the resource-operation model.

```
1  namespace Example {
2    class Queue {
3      [Resources("list: get addFirst addLast"]
4      protected List<int> _items = new List<int>();
5
6      [Semantics("list.addLast")]
7      public virtual void Add(int i) {
8        _items.Add(i);
9      }
10
11     [Semantics("list.get")]
12     public int Get() {
13       return _items[0];
14     }
15   }
16
17   class Stack : Queue {
18     [Semantics("list.addFirst")]
19     public override void Add(int i) {
20       _items.Insert(0, i);
21     }
22   }
23
24   class Test {
25     public static void Main() {
26       Stack sub = new Stack();
27       Queue super = sub;
28       super.Add(0);
29     }
30   }
31 }
```

Listing 8.13: Example of subtyping with behavioral descriptions

Listing 8.13 is based on the example from Section 8.1, but now with a behavioral description added. This example shows that although `Stack` is a subtype of `Queue`, they are not semantically equivalent. This is indicated by the different behavioral descriptions of the `Add` methods. By analyzing the custom attributes, we could determine that `Stack` exhibits different behavior than `Queue` and is therefore not a valid behavioral subtype.

However, there are two problems with the example. Suppose the `list` resource had a more limited vocabulary: `add get`. Both `Add` methods would be described with `list.add`, although they are clearly not exhibiting the same behavior. The second problem

is the fact that the context is not taken into consideration. If all that is needed is a list in which we can store integers and it does not matter in which order, then there is no difference in the behavior of both classes for the caller.

The problems with the example are the result of the resource-operation model being too abstract and too generic to represent the behavior of a program. This results in a number of general problems when attempting to describe the behavior with the resource-operation model.

Lack of language constructions. The resource-operation model is too generic, which means that it is lacking any specific functionality to describe the behavior of source code. Constructions used in other behavior describing formats, such pre-conditions, post-conditions, invariants, must be described by constraints. Although this could be possible, it makes the description unnecessarily difficult and large.

Different scope for operations. Another problem with the language model being too generic is that everything has to be written in terms of operations on a resource. This leads to very different operations being performed on the same resource. For example, `list.add` represents a concrete method that adds an element to the list whereas `list.notEmpty` represents an abstract requirement that the list is not empty.

No constraints on resources or operations. The lack of constraints on resources makes it very difficult to model properties of resources. For example, the requirement that a resource has to be an integer and should be greater than zero. The lack of constraints on operations makes it very difficult to model the relations between different operations. For example, if a method in the supertype performs no actions and the method in the subtype performs a read operation on a resource, then there is no way to tell whether the method in the subtype exhibits the same behavior.

The resource-operation model in its current form is not a proper solution for describing the behavior of a program. The problems encountered when trying to describe the behavior with resource-operation model mean that comparing different behaviors is going to be a very difficult task. This is indicated by the fact that the model appears to incompatible with the definition for behavioral subtype relations in Section 8.1.1.

The resource-operation model could be adapted to better describe the behavior of a program, but this would require extending the resource-operation model to the point that it is no longer the abstract model it was designed to be. A better solution would be to use another format to describe the behavior of the source code instead of the resource-operation model.

8.4.2 Extending the filter analysis

An alternative to the full representation and analysis of the behavior of the program could be to extend the existing filter analysis. To allow the filter analysis to reason

about the impact of the filter on the behavior of the original method, the behavior of both the extending method and the filter is needed.

The work by Tom Staijen [50] was already mentioned when discussing custom attributes. His work focused on describing the behavior of meta-filters by annotating the called methods with custom attributes that describe the behavior of the method. This is only performed on methods called through a meta-filter, but this could be extended to methods called by other filters.

By describing the behavior of other methods called through a filter and retrieving that information, SECRET could incorporate this information and reason about potential behavioral conflicts introduced by the called method. To simplify the reasoning process and the implementation of this approach, the behavioral description of a method uses the same resource-operation model used by SECRET. This limits the behavioral descriptions to the resources defined by SECRET.

As an example, we modify the method extending example given in Section 8.3.1. The concern still applies a `Before` filter to any calls to the `Test` method in class `A`. The method now called before `A::Test` is the `Modify` method in class `C`.

The new method `Modify` has a custom attribute that describes the actions performed by the method that are relevant for the filter analysis. In this case, the only performed action that is relevant is described by `arg.write`, which indicates that the arguments of the message are modified.

```
namespace Test {
  class A {
    public void Test(B b) {
    }
  }

  public class C {
    [Semantics("arg.write")]
    public void Modify (JoinPointContext jpc) {
      for (short s = 0; s < jpc.ArgumentCount; s++) {
        if (jpc.GetArgumentType(s) is B) {
          B b = (B) jpc.GetArgumentValue(s);
          // Modify b
          b.Modify();
        }
      }
    }
  }
}
```

Listing 8.14: Example of a class with behavioral description

```
concern MethodTesting in Test
{
  filtermodule ExtendedMethod
  {
    externals
      c: Test.C;
    inputfilters
```

```
8         log: Before = { [*.Test] c.Modify }
9     }
10
11    superimposition
12    {
13      selectors
14        subjects = { C | isClassWithName(C, 'Test.A') };
15      filtermodules
16        subjects <- ExtendedMethod;
17    }
18  }
```

Listing 8.15: Before filter extending Test method in class A

The modified example can be used to help SECRET generate a more informative sequence of resource-operation pairs, allowing it to better reason about potential behavioral conflicts introduced by the `Modify` method.

```
1  Test selector: arg.read, arg.write, message.dispatch, return.read,
       message.return
2    + selector: arg.read, message.dispatch, return.read, message.return
```

The first sequence in the SECRET output now indicates that the arguments of the message are modified before the orginal method is called. This means that the original method is called with different arguments than is expected by the calling code. This could indicate a potential conflict in the expected behavior and the observed behavior of the call made to `A:Test`.

Without more information, it is impossible for SECRET to determine with certainty that the behavior of the `Modify` conflicts with the behavior of `Test`. However, a warning could be issued to the developer informing him of the potential problem.

Chapter 9

Conclusions

This chapter discusses the results of the research presented in this thesis. This is done in two sections: first, we look at the improved compilation process. Second, we look at extending the behavioral analysis by Compose⋆. In both sections, we evaluate our findings and discuss possible future work.

9.1 Improved compilation process

9.1.1 Evaluation

We evaluate the modified compilation process by looking at the three stages of its development process.

Theoretical solution. The modified compilation process was in theory a viable alternative for the original compilation process. It supports signature modification while using the standard .NET compilers. Furthermore, it removed the language dependencies disadvantage. One previous disadvantage could not be solved: there is still no support for self-referencing types. The modified compilation process introduced one disadvantage: it depends on Visual Studio, making Visual Studio necessary to compile with Compose⋆/.NET. We felt that this disadvantage was outweighted by the advantages of using Visual Studio.

Implementation. The implementation of the improved compilation process resulted in no major errors. A few minor problems were encounted that are either manageable (some small language dependencies) or not yet implemented (modifications to the Compose⋆/.NET project and the support for creatiing PDB files).

Testing The backwards compatibility tests were only aimed at C#, no tests were performed to test the support for J#. The C# tests revealed some minor problems with global delegates and nested classes. The problem with nested classes was identified to be a lack of support in the new code generating task and was fixed.

The testing of the support for signature modification revealed a number of problems:

- Errors or warnings do not inform the developer that they are caused by a composition filter.
- Conditional signature modification is not supported by Compose★/.NET. All reasoning about conditional signature modification was therefore theoretical.

9.1.2 Future work

The tests performed on the improved compilation process have resulted in a number of recommendations.

- If an error or warning at compile-time is caused by a composition filter, then the message has to be extended with information about the filter.
- Conditional signature modification has to be implemented. The solution for conditional signature modification will also be applied to signature reduction.
- Improvements to the performed tests are needed. These improvements are twofold:
  - Extend the backwards compatibility tests to include all programming languages supported by Compose★/.NET (most notably J#).
  - Automate both the backwards compatibility and the signature modification testing.
- The current version of Compose★/.NET targets the .NET 2.0 Framework. Support for .NET 3.0 and .NET 3.5 and the accompanying Visual Studio 2008 could be implemented into a new version of Compose★/.NET. The improved compilation process could also be applied to the old .NET 1.1 Framework and Visual Studio 2003.

Finally, although the compilation process was improved in many areas, there are still some disadvantages to the current approach. More research into other solutions could result in a compilation process that addresses the currently existing disadvantages.

9.2 Behavioral subtyping

We evaluate our attempt to extend the existing behavioral analysis in Compose★ to include the analysis of the behavior of the original source code.

9.2.1 Evaluation

Our approach to a full behavioral analyis was to extend the existing behavioral analysis of Compose⋆. This existing analysis is the filter analysis by SECRET. To reason about the behavior of the filters, SECRET uses the resource-operation model to describe the behavior. Therefore, our extension of the analysis would also use the resource-operation model.

Extending the behavior analysis introduces a number of steps. First, we need to determine the behavior of the original source code. This behavior has to be described in the resource-operation model. Next, the found behavior has to be combined with the behavior exhibited by the composition filters to create the behavior of the new subtypes. Finally, the behavior of the new subtypes, as created by the composition filters, has to be compared with the behavior of the orginal types to determine behavioral conflicts.

There are several valid approaches to the first step (determining the behavior). Examples were shown, such as Larch/C++. The next step is describing this behavior in the the resource-operation model. Unfortunately, the resource-operation model was found to be too generic and too abstract to properly describe all behavior. Extending the resource-operation model could resolve the encountered issues, but better alternative behavioral description formats are available.

With the resource-operation model unable to describe and reason about the behavior of the source code in its current form, we have not looked into approaches for the final steps of the full behavioral analysis process. It is not very useful to look at the final steps of the analysis process, since they are likely to change when an alternative to the current resource-operation model would be used to describe the behavior.

9.2.2 Future work

The full behavioral analysis approach was found to unpractical to implement. We have provided an alternative that extends the existing behavioral analysis in a limited way. This extension should be easy to implement, since it does not extend the resource-operation model in use. However, further work is required to determine whether the extending of the behavioral analysis is practical and an asset to Compose⋆.

Bibliography

[1] Ada. Ada for the web, 1996. URL http://www.acm.org/sigada/wg/web_ada/.

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

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

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

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

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

[7] A. J. de Roo. Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis, University of Twente, The Netherlands, Mar. 2007.

[8] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. 1997.

[9] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, Oct. 2006.

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

[11] Ecma. Eiffel: Analysis, design and programming language. 2006. URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf.

[12] Eiffel Software. Building bug-free o-o software: An introduction to design by contract(tm). 2004. URL http://archive.eiffel.com/doc/manuals/technology/contract/.

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

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

[15] 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.

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

[17] J. Guttag, K. J. A. M. J.J. Horning, with S.J. Garland, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

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

[19] M. F. H. Hendriks. Construction visualization of dispatch graphs for Compose*. Master's thesis, University of Twente, The Netherlands, 2007.

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

[21] R. L. R. Huisman. Debugging Composition Filters. Master's thesis, University of Twente, The Netherlands, Feb. 2007.

[22] S. H. G. Huttenhuis. Using aspect-oriented programming to solve problems of design pattern implementations. Master's thesis, University of Twente, The Netherlands, 2006. Nov.

[23] E. International. Common language infrastructure (CLI). Standard ECMA-335, ECMA International, 2002. URL http://www.ecma-international.org/publications/files/ecma-st/Ecma-335.pdf.

[24] R. Jongeling. New signature expansion process for compose*. 2008.

[25] R. Jongeling. Signature modification for Compose*.net. Master's thesis, University of Twente, The Netherlands, 2008.

[26] Jython. Jython homepage. URL http://www.jython.org/.

[27] 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.

[28] 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`.

[29] Larch. Larch home page. URL `http://www.sds.lcs.mit.edu/spd/larch/`.

[30] G. T. Leavens. Larch/c++: An interface specification language for c++. 1997.

[31] B. H. Liskov and J. M. Wing. Behavioral subtyping using invariants and constraints. 1999.

[32] Microsoft Corporation. Overview of the .NET framework. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp`.

[33] Microsoft Corporation. What is the common language specification. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp`.

[34] Microsoft Corporation. .NET compact framework - technology overview. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/overview/default.aspx`.

[35] Microsoft Corporation. What's is .NET? Technical report, Microsoft Corporation, 2005. URL `http://www.microsoft.com/net/basics.mspx`.

[36] Microsoft Corporation. Rotor. 2002. URL `http://research.microsoft.com/collaboration/university/europe/RFP/rotor/default.aspx`.

[37] MSDN. Automation object model chart. 2008. URL `http://msdn.microsoft.com/en-us/library/za2b25t3(VS.80).aspx`.

[38] MSDN. Using the codedom. 2008. URL `http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx`.

[39] MSDN. Envdte namespace. 2008. URL `http://msdn.microsoft.com/en-us/library/envdte(VS.80).aspx`.

[40] MSDN. System.codedom namespace. 2008. URL `http://msdn.microsoft.com/en-us/library/system.codedom.aspx`.

[41] MSDN. Fixing 'application is busy' and 'call was rejected by callee' errors. 2008. URL `http://msdn.microsoft.com/en-us/library/ms228772(VS.80).aspx`.

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

[43] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[44] T. Parr. Antlr parser generator. 2007. URL `http://www.antlr.org/`.

[45] 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.

[46] 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.

[47] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2002. ISBN 0-7356-1376-1.

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

[49] D. R. Spenkelink. Compose* incremental. Master's thesis, University of Twente, The Netherlands, Dec. 2006.

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

[51] D. Stutz. The Microsoft shared source CLI implementation. *MSDN Magazine*, 17 (7), July 2002.

[52] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[53] J. W. te Winkel. Bringing Composition Filters to C. Master's thesis, University of Twente, The Netherlands, Dec. 2006.

[54] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society, Series 2, 42*, pages 230–265, 1936.

[55] University of Linz. Coco/r. 2007. URL `http://www.ssw.uni-linz.ac.at/coco/`.

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

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

[58] D. Watkins. Handling language interoperability with the Microsoft .NET framework. Technical report, Monash Univeristy, Oct. 2000. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/interopdotnet.asp`.

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

[60] 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`.

Appendix A

Results signature modification testing

Stand-alone method scenarios

Test: Stand-alone signature expansion with a private method

Description:
A stand-alone private method is added to a type and the method is called through an object of this type.

Expected behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. No runtime test possible.

Actual behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. No runtime test possible.

Desired behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. This error message to the developer should be extended to include information that the method is added by a filter. No runtime test is possible.

Test: Stand-alone signature expansion with a protected method

Description:
A stand-alone protected method is added to a type and the method is called through an object of this type.

Expected behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. No runtime test possible.

Actual behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. No runtime test possible.

Desired behavior:
The compilation fails with the message that the called method is inaccessible due to its protection level. This error message to the developer should be extended to include information that the method is added by a filter. No runtime test is possible.

Test: Stand-alone signature expansion with a static method

Description:
A stand-alone static method is added to a type and the method is called statically.

Expected behavior:
The compilation succeeds and at runtime the static method call succeeds.

Actual behavior:
The compilation succeeds and at runtime the static method call succeeds.

Desired behavior:
The actual behavior matches the desired behavior.

Test: Stand-alone signature expansion with an abstract method

Description:
A stand-alone abstract method is added to a non-abstract type and the method is called through an object of this type.

Expected behavior:
The compilation fails with the message that the abstract method is contained in a non-abstract class. No runtime test possible.

Actual behavior:
The compilation fails with the message that the abstract method is contained in a non-abstract class. No runtime test possible.

Desired behavior:
The compilation fails with the message that the abstract method is contained in a non-abstract class. This error message to the developer should be extended to include information that the method is added by a filter. No runtime test is possible.

Test: Signature reduction of a generic type

Description:
See Listing 7.3: the `SetData` method is removed from the generic `GenericTypeTest` type. The method is called through an object of this type as is defined in `GenericsExample`.

Expected behavior:
The compilation fails with the message that `GenericTypeTest` does not contain the `SetData` method. No runtime test possible.

Actual behavior:
The compilation fails with the message that `GenericTypeTest` does not contain the `SetData` method. No runtime test possible.

Desired behavior:
The compilation fails with the message that `GenericTypeTest` does not contain the `SetData` method. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test possible.

Test: Signature expansion of a non-generic type with a generic method

Description:
See Listing 7.3: the `Swap` method is added to the `GenericMethodTest` type. The method is called statically as is defined in `GenericsExample`.

Expected behavior:
The compilation succeeds and at runtime the static method call succeeds.

Actual behavior:
The compilation succeeds and at runtime the static method call succeeds.

Desired behavior:
The actual behavior matches the desired behavior.

Test: Signature reduction of a non-generic type with a generic method

Description:
See Listing 7.3: the `Swap` method is removed from the `GenericMethodTest` type. The method is called statically as is defined in `GenericsExample`.

Expected behavior:
The compilation fails with the message that `GenericMethodTest` does not contain the `Swap` method. No runtime test possible.

Actual behavior:
The compilation fails with the message that `GenericMethodTest` does not contain the `Swap` method. No runtime test possible.

Desired behavior:
The compilation fails with the message that `GenericMethodTest` does not contain the `Swap` method. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test possible.

Methods that are part of inheritance

Test: signature reduction of a superclass

Description:
The `Test` method is present in the signature of `SuperClass`, but not in `SubClass`. The `Test` method is then removed from `SuperClass` and the method is called through an object of `SubClass` that should no longer inherit the method.

Expected behavior:
The compilation fails with an error that the `SubClass` does not contain a definition for the `Test` method. No runtime test is possible.

Actual behavior:
The compilation fails with an error that the `SubClass` does not contain a definition for the `Test` method. No runtime test is possible.

Desired behavior:
The compilation fails with an error that the `SubClass` does not contain a definition for the `Test` method. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test is possible.

Test: signature expansion of a subclass

Description:
The `Test` method is not present in the signature of both classes. The `Test` method is then added to `SubClass` and the method is called through an object of `SuperClass`.

Expected behavior:
The compilation fails with an error that the `SuperClass` does not contain a definition for the `Test` method. No runtime test is possible.

Actual behavior:
The compilation fails with an error that the `SuperClass` does not contain a definition for the `Test` method. No runtime test is possible.

Desired behavior:
The actual behavior matches the desired behavior.

Test: Signature reduction of a subclass with an inherited method

Description:
The `Test` method is present in the signature of both classes. The `Test` method is then removed from `SubClass` and the method is called through an object of `SubClass` which should result in a call to the method in `SuperClass`.

Expected behavior:
The compilation succeeds and at runtime the method call calls the method defined in `SuperClass`.

Actual behavior:
The compilation succeeds and at runtime the method call calls the method defined in `SuperClass`.

Desired behavior:
The actual behavior matches the desired behavior.

Test: Signature reduction of a subclass inheriting an abstract method

Description:
The `Test` method is present in the signature of both classes, but both `Test` and `SuperClass` are abstract. The `Test` method is then removed from `SubClass` and the method is called through an object of `SubClass`.

Expected behavior:
The compilation fails with an error that the `SubClass` does not implement the abstract `Test` method inherited from `SuperClass`. No runtime test is possible.

Actual behavior:
The compilation fails with an error that the `SubClass` does not implement the abstract `Test` method inherited from `SuperClass`. No runtime test is possible.

Desired behavior:
The compilation fails with an error that the `SubClass` does not implement the abstract `Test` method inherited from `SuperClass`. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test is possible.

Test: Signature reduction of a subclass inheriting method from an interface

Description:
The `Test` method is present in the signature of `SubClass` and is inherited from an interface. The `Test` method is then removed from `SubClass` and the method is called through an object of `SubClass`.

Expected behavior:
The compilation fails with an error that the `SubClass` does not implement interface member `Test`. No runtime test is possible.

Actual behavior:
The compilation fails with an error that the `SubClass` does not implement interface member `Test`. No runtime test is possible.

Desired behavior:
The compilation fails with an error that the `SubClass` does not implement interface member `Test`. No runtime test is possible. This error message to the developer should be extended to include information that the method is removed by a filter. No runtime test is possible.

Hiding methods

Test: Signature expansion of a subclass with an hiding method

Description:
The `Test` method is present in the signature of `SuperClass`. The `Test` method is then added to `SubClass` hiding the method in `SuperClass`. The method is called through an object of `SuperClass`.

Expected behavior:
The compilation succeeds and at runtime the method call calls the `Test` method defined in `SuperClass`.

Actual behavior:
The compilation succeeds and at runtime the method call calls the `Test` method defined in `SuperClass`.

Desired behavior:
The actual behavior matches the desired behavior.

Test: Signature expansion of a subclass with an implicitly hiding method

Description:
The `Test` method is present in the signature of `SuperClass`. The `Test` method is then added to `SubClass`, implicitly hiding the method in `SuperClass`. The method is called through an object of `SuperClass`.

Expected behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SuperClass`.

Actual behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SuperClass`.

Desired behavior:
The compilation succeeds, but gives a warning that the `Test` method in `SubClass` hides an inherited member. This warning message to the developer should be extended to include information that the hiding method is added by a filter. At runtime, the method call calls the method defined in `SuperClass`.

Test: Signature expansion of a superclass with an hidden method

Description:
The `Test` method is present in the signature of `SubClass`. The `Test` method is then added to the `SuperClass`, leaving the hiding method in `SubClass`. The method is called through an object of `SubClass`.

Expected behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SubClass`.

Actual behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SubClass`.

Desired behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. This warning message to the developer should be extended to include information that the hidden method is added to `SuperClass` by a filter. At runtime the method call calls the method in the `SubClass`.

Test: Signature expansion of a superclass with an hidden method

Description:
The `Test` method is present in the signature of `SubClass`. The `Test` method is then added to the `SuperClass`, leaving the hiding method in `SubClass`. The method is called through an object of `SuperClass`.

Expected behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SuperClass`.

Actual behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. At runtime, the method call calls the method defined in `SuperClass`.

Desired behavior:
The compilation succeeds, but gives a warning that the method in `SubClass` hides an inherited member. This warning message to the developer should be extended to include information that the hidden method is added to `SuperClass` by a filter. At runtime the method call calls the method in the `SuperClass`.

Test: Signature reduction of a superclass with an hidden method

Description:
The `Test` method is present in the signature of both class. The `Test` method is then removed from the `SuperClass`, leaving the hiding method in `SubClass`. The method is called through an object of `SubClass`.

Expected behavior:
The compilation succeeds, but gives a warning that there is no method to hide. At runtime the method call calls the method in the `SubClass`.

Actual behavior:
The compilation succeeds, but gives a warning that there is no method to hide. At runtime the method call calls the method in the `SubClass`.

Desired behavior:
The compilation succeeds, but gives a warning that there is no method to hide. This warning message to the developer should be extended to include information that the method is removed from `SuperClass` by a filter. At runtime the method call calls the method in the `SubClass`.

Conditional signature modification

Test: Conditional removal of a hiding method from a subtype

Description:
The `Test` method is present in the `SuperClass` and is hidden by `Test` in `SubClass`. The `Test` method is then conditionally removed from `SubClass`. A `SubClass` object is cast to `SuperClass` and the `Test` method is called on the `SuperClass` object twice; once with the method in place and once with the method removed.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. At runtime, the framework will simply call the `Test` method as defined in `SuperClass` since it recognizes that the `Test` method is never inherited by `SubClass`. The conditional removal of the `Test` method from the signature of `SubClass` will have no effect on the call.

Actual behavior:
The compilation fails, since the method is removed permanently at compile-time by Compose* and any calls to the method on a `SuperClass` object will be recognized by the .NET compiler as being invalid.

Desired behavior:
The compilation succeeds, but a warning message should be issued to the developer informing that the resulting program might fail at runtime due to the missing method. The call to the removed method should fail at runtime, even if the call is actually to the existing method in the subtype. The method was removed from the signature of the object's type and any calls to it should therefore fail. The error message that is shown when the program fails, should explain that the failure is due to an invalid call to a method that has been removed by a composition filter.

Test: Conditional removal of a hiding method from a subtype (2)

Description:
The `Test` method is present in the `SuperClass` and is hidden by `Test` in `SubClass`. The `Test` method is then conditionally removed from `SubClass`. A `SubClass` object is cast to `SuperClass` and the `Test` method is called on the `SubClass` object twice; once with the method in place and once with the method removed.

Expected behavior:
The compilation succeeds, since at compile-time the method is present in the code and any calls to the method are legal. At runtime, the framework will simply call the `Test` method as defined in `SubClass` when the method is present. The call to the removed `Test` should result in a call to the `Test` method in the `SuperClass`.

Actual behavior:
The compilation fails, since the method is removed permanently at compile-time by Compose⋆ and any calls to the method on a `SuperClass` object will be recognized by the .NET compiler as being invalid.

Desired behavior:
The expected behavior is the desired behavior.