

Towards Safe Advice:

Semantic Analysis of Advice Types in Compose*

T. Staijen

April 2005

Software Engineering group,
Electrical Engineering, Mathematics
and Computer Science,
University of Twente

Graduation committee:

Prof. Dr. Ir. M. Aksit
Dr. Ir. L.M.J. Bergmans
Ir. P.E.A. Dürr

Abstract

The use of Aspect Orientation has the potential to increase the separation of concerns and to identify crosscutting aspects. The Composition filters approach to AOP is based on filtering of messages between objects. This allows for more flexible and reusable aspects, in comparison to other approaches like AspectJ.

A powerful characteristic of Compose*, the implementation of the Composition Filters model for the .Net platform, is the possibility to analyze how filters will interact or conflict with one another.

One of the filters used in the CF model is the Meta-filter, which reifies a message and sends it to a first-class object that abstracts the communication among objects, a so called Advice Type (ACT).

Since the Meta-filter enables a programmer to do practically anything with a message, it is no longer possible to reason with the behavior of the filter. This means that conflicts that might occur between filters can no longer be detected.

In this thesis the definition of ACTs is enhanced and a solution is presented to specify the behavior of the ACT so that the system can reason about the definition of the meta-filter. Also, the run-time filtering system is extended to support the definition of ACTs.

Although the solution is implemented for Compose*, its semantic reasoning capabilities could be used for any Aspect Oriented approach that uses turing-complete advice.

Acknowledgements

Writing a thesis is quite a bit of work. Especially when you start writing at a late stage and you find out that writing helps you seeing everything you overlooked during your research. After all it wasn't such a late stage at all; it just took very long to get there. Finally being able to finish up, I'd like to thank some people for their assistance in this effort.

First of all my supervisor, Lodewijk Bergmans, for his guidance and never-ending inspiration. Without him I might never have realized the magnificence of Aspect Orientation Software Development.

Also, I wish to thank the members of my graduation committee for their remarks and suggestions. I appreciate their dedication and giving up their time in attending all those meetings and supporting me.

Some friends I'd like to mention in person for bringing some fun to all this work: Pascal Dürr, Frederik Holljen and Raymond Bosman. I'll never forget the hours we spent together in the lab.

Last but certainly not least, I thank my parents and my brother for their support and interest, but mostly for their patience...

Contents

Abstract	i
Acknowledgements	iii
1 Introduction and Background	1
1.1 Introduction to Aspect-Oriented Software Development	1
1.1.1 The object-oriented approach to software development	3
1.1.2 Problems with the object-oriented approach	3
1.1.3 Example	4
1.1.4 The AOP solution	5
1.1.5 AOP composition	5
1.1.6 Aspect weaving	6
Source code weaving	6
Intermediate Language weaving	7
Adapting the Virtual Machine	8
1.1.7 AOP approaches	8
The AspectJ approach	8
The Hyperspaces approach	10
1.2 Composition Filters	11
1.2.1 The Composition Filters approach	11
The superimposition mechanism	13
1.2.2 Evolution of Composition Filters	14

2	Compose*	17
2.1	Introduction	17
2.2	Overview of the .NET architecture	17
2.2.1	Features of the Common Language Runtime	18
2.2.2	The .NET Framework class library	19
2.2.3	Standardization	20
2.2.4	A comparison between the .NET CLR and the Java VM	21
2.3	Features explicit to Compose*	23
2.4	Demonstrating example	24
2.4.1	The object-oriented design	24
2.4.2	Completing the Pacman example	26
2.5	Architecture	28
	Master	28
	TYM (TYpe Manager pass one)	28
	SUPRE (SUperimpositon PREprocessing)	30
	REXREF (Resolve EXternal REferences)	30
	SANE (Superimposition ANalysis Engine)	30
	FILTH (FILTer composition & cHecking)	30
	FIRE (FIlter Reasoning Engine)	30
	SIGN (SIGNature GeNeration)	31
	CORE (COde geneREation)	31
	SECRET (SEmantiC Reasoning Tool)	31
	CONE (COde geNEration	32
	TYM (TYpe Manager pass two)	32
2.5.1	The Repository	32
2.5.2	The Compose* Runtime environment	33
	ILICIT (InterCeption InserTer)	34
	Interception Handler	34

Filter Composition Constraints	34
SuperImposter	34
FLIRT (FiLter InteRpreTer)	35
3 Problem Statement	37
3.1 A Meta-filter example: Jukebox system	37
3.2 The aspect interaction problem	39
3.3 Aspect interaction in Compose*	40
3.3.1 Aspect interference and Advice Types	41
3.4 The Goal	41
4 Analysis of Advice Types	43
4.1 Message property manipulation	43
4.2 Message execution manipulation	44
4.2.1 Resume	44
4.2.2 Proceed	44
4.2.3 Return	45
4.2.4 Respond	45
4.2.5 Send	46
4.2.6 Combining execution operations	46
5 Solution model	49
5.1 Defining ACT semantics	49
5.2 Resource-model instantiation	50
5.2.1 Meta filter	50
Conceptual resources	51
5.2.2 Dispatch filter	52
5.2.3 Error filter	52
5.3 Modifying the analysis engine	52

5.4	Conflict detection	54
5.4.1	Respond twice	54
5.4.2	Proceed and error	54
5.4.3	Writing after returning	55
5.4.4	Playing a song without paying it	55
6	Design and Implementation	57
6.1	Specification of semantics using annotations	58
6.2	Collecting and storing annotations	59
6.3	Initialization of the Reasoning Engine	60
6.4	Implementation of the reasoning engine	61
6.4.1	Reporting	63
6.4.2	Coupling with FIRE	63
6.5	Implementation of the Meta-filter run-time	65
6.5.1	Concurrency	65
6.5.2	Message Queue	67
6.5.3	Including MetaActions in the run-time	67
7	Conclusions	73
7.1	Related work	73
7.2	Discussion & limitations	73
7.3	Conclusion	74
7.4	Future work	75
7.4.1	Alphabet checking	75
7.4.2	Respond a proxy	75
7.4.3	Filter impossible executions	75
7.4.4	Annotate dispatched methods	76
7.4.5	Comparing filtermodule-orders	76

Appendices	78
A SECRET Configuration file	79

Chapter 1

Introduction and Background

The first two chapters have been written by six MSc. students at the University of Twente. These serve as a general introduction into Compose* and the underlying techniques. The chapters are used in the master theses of these authors:

Compilation and Type-Safety in the Compose* .NET environment.

Frederik J. B. Holljen

Detecting semantic conflicts between aspects.

Pascal E. A. Dürr

Automated Reasoning about Composition Filters.

Raymond Bosman

Superimposition in the Composition Filters model.

Christian A. Vinkes

Towards Safe Advice: Semantic Analysis of Advice Types in Compose*.

Tom Staijen

Performing transformations on .NET Intermediate Language code.

Sverre R. Boschman

1.1 Introduction to Aspect-Oriented Software Development

Ten years ago the dominant programming language paradigm was imperative programming. This paradigm is characterized by the use of commands that update variables. Most popular are the Algol-like languages, such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented languages. Figure 1.1 summarizes the dates and ancestry of several important languages [31].

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

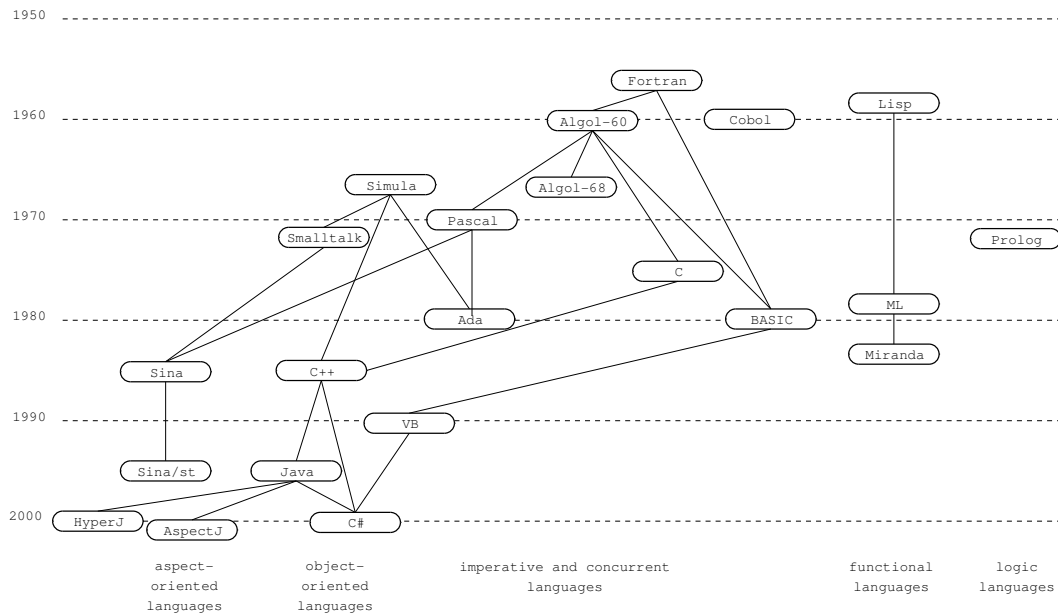


Figure 1.1: Dates and ancestry of several important languages

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 [31].

Object-oriented languages are related closely to the imperative programming languages. Most Object-Oriented Programming (OOP) languages are extensions of imperative programming, based on classes and objects. An object is a variable that may be accessed only through operations associated with it. Although the concept appeared in the seventies, it took 20 years to become popular. The most well known object-oriented languages are C++, Java and Smalltalk [31].

Aspect-Oriented Programming (AOP) is a paradigm that solves the problem of crosscutting concern. We recognize two forms of crosscutting: code tangling and code scattering. Code tangling occurs when multiple concerns are implemented within the same system element. Code scattering accrues when a concern results in the implementation of duplicate or different code that is distributed across multiple system elements [20, 21]. AOP introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of aspect-oriented languages are: Sina, AspectJ and HyperJ.

AOP is commonly used in combination with OOP. The following sections discuss the OOP paradigm, the problems that may rise with OOP, and how AOP can help to solve these problems. Finally, we look at three particular AOP implementations in more detail.

1.1.1 The object-oriented approach to software development

The following discussion is derived from Gamma et al.[14]:

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client.

Requests are the only way to get an object to execute an operation. Operations are the only way to change an object's internal data. Because of these restrictions, the object's internal state is said to be encapsulated; it cannot be accessed directly, and its representation is invisible from outside the object.

A type is a name used to denote a particular interface. An object may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a subtype of another if its interface contains the interface of the supertype. Often we speak of a subtype inheriting the interface of its supertype.

An object's implementation is defined by its class. The class specifies the object's internal data and representation and defines the operations the object can perform.

Objects are created by instantiating a class. The object is said to be an instance of the class. The process of instantiating a class allocates storage for the object's internal data and associates the operations with these data. Many similar instances of an object can be created by repeatedly instantiating a class.

The main advantage of the OOP approach over imperative programming is modularity. A system is decomposed into multiple objects. Each object 'lives' independent of each other in the system. The improvement of re-usability, flexibility, performance and evolution are influenced by the factor of decomposition and the used object-oriented techniques (inheritance, polymorphism, overloading, implementation).

Despite the improvement introduced by object-oriented programming, there are still problems which cannot easily be solved using the object-oriented model. These problems are discussed in the next section.

1.1.2 Problems with the object-oriented approach

Ideally, an object should be a unit with as little knowledge as possible of its surrounding environment. The surrounding environment is composed of the other objects in the system and the only knowledge the object has about those other objects is the information they expose through their interface. Similarly, the only thing the environment should know about the object is what it exposes through its interface. This is accomplished by encapsulation. The resulting object should ideally implement one concern of the system. A concern is a requirement, or some piece of functionality originating from the requirements, which has been implemented in a code structure. By realizing all the concern, the system should be able to accomplish the goals it has

been designed to achieve. However, while designing the system, one cannot break the rules imposed by the design methodology used. When using the object-oriented approach for example, programmers are bound by the limitations of that approach. As we will see, the object-oriented approach does not hold up when pieces of functionality are required to be implemented across several objects [13]. As [24] et al. point out, in most cases formalisms such as programming languages and design notations only provide one prevalent means of decomposing software (they only support one dimension of concern). This causes problems when formalisms for the same software use different dimensions of concern. For example, requirements are often specified by a function or feature while object-oriented design and code is decomposed using classes. This creates a conceptual mismatch and requires developers to switch constantly between different representations of the same concept.

1.1.3 Example

Consider an application containing an object A, which adds two integers. The application also has a `LogWriter` object to write messages about the program execution to a log file. This is done using a method called `write()`. Furthermore, suppose object A needs to write the result of the addition to the log file. The definition of object A might look something like listing 1.1.

```
1 public class A {
2     private LogWriter log;
3     public int a, b;
4
5     A() {
6         log = new LogWriter();
7     }
8
9     public void addTwoIntegers () {
10        private int result;
11
12        result = a + b;
13        log.write(calculation performed);
14    }
15 }
```

Listing 1.1: Modeling logging without aspects

By adding the logging code to class A, the class the concerns (it not only implements its “own” concern, but also handles the requirements of a second concern). Crosscutting is the situation where a system requirement is met by placing code into different objects throughout the system [16]. This results in tangled code in the system; the implementation of the concerns is now scattered throughout the system. Tangled code creates the following problems:

the code is difficult to change: If the interface of the logging object changes, changes will need to be made throughout the system to adjust to the new interface.

the code is harder to reuse: In order to reuse object A in another system, it is necessary to either remove the logging code or reuse the logging object in the new system.

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

It is clear that problems described will only become worse if the systems evolves with additional classes using logging.

1.1.4 The AOP solution

To solve the problems with the OO approach, several techniques are being researched that attempt to increase the expressiveness of the OO paradigm [12]. Such techniques are known as Post-Object Programming (POP) mechanism [12]. Aspect-oriented programming is one such POP technology. AOP allows all concerns that must be implemented in a system, to be clearly expressed in a way that is not possible using the object-oriented approach. A special syntax is used to specify aspects and the way in which are combined with regular objects. However, AOP is not a replacement but an extension of OOP [2], i.e. objects can still be used. The fundamental goals of AOP are twofold [16]: first of all, to provide a mechanism for the description of concerns that crosscut other components. And secondly to use this description to allow for the separation of concerns.

```
1 aspect Logging {
2     LogWriter log = new LogWriter();
3     pointcut log(): call(A.addTwoIntegers());
4
5     after(): log() {
6         log.write(calculation performed);
7     }
8 }
```

Listing 1.2: Modeling logging using aspects

Listing 1.2 creates a new aspect which executes the logging code after each call to `addTwoIntegers`. Line 3 specifies the pointcut, i.e. where to execute the aspect code; in this case, when the `addTwoIntegers` method is called on an object of class `A`. Line 5 specifies when to execute the code; in this case, after the completion of the `addTwoIntegers` method. Using aspects has several advantages over the previous code [12][2]:

the crosscutting concern is explicitly captured: Instead of being embedded in the code of other objects, aspects are now made visible and specified outside the objects.

the evolution of the code is simplified: A component can be changed without interfering with other components or aspects in the system.

an encapsulated concern can be reused: Both the aspect and the component are now fully separated and can be reused in other systems.

the ability to include / exclude functionality: Since aspects are separated from the components, adding or excluding them is a lot easier.

1.1.5 AOP composition

A program which uses AOP techniques, can be thought of being composed of two parts:

1. The component part consisting of the base program. The language used to write this part is also called the component language.
2. The part consisting of aspects. The language used to write this part is also called the aspect language. The aspect language can differ from the component language.

This model, also called the asymmetric approach is followed by AspectJ (covered in more detail in the next section). It is, however, not mandatory to have two parts; for example, HyperJ is not clearly separated into a component- and aspect part. This is called the symmetric approach. According to [2], a successful separation of concerns can be characterized by the following adjectives:

Simultaneous: Different decompositions need to be able to coexist.

Self-contained: To make sure each module can be understood in isolation, it should specify its dependencies.

Symmetric: To assure that modules encapsulating different kinds of concerns can be composed together in a flexible way, there should be no distinction in form between them.

Spontaneous: As new concerns appear during the software life cycle, it should be possible to identify and encapsulate them.

1.1.6 Aspect weaving

The integration of components and aspects is called *aspect weaving*. There are three locations where composition mechanisms can be applied in order to support aspect weaving. The first and second approach rely on adding behavior in the program either through weaving the aspect through the developers source code or a directly into the target language. The target language can be an Intermediate-Language (IL) (such as Java Byte code, MSIL, etc) or machine code. The remainder of this chapter considers only IL language targets. The third approach relies on adapting the interpreter. Each method is explained briefly in the following sections.

Source code weaving

The source code weaver combines the original source with aspect code. Therefore this weaver interpreters the defined aspects and generates, together with the original source, 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 (of course the output depends 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 language of the native compiler.

- Aspect and original source optimization. First the aspects are woven through the source code and hereafter compiled by the native compiler. The produced target language has all the benefits of the native compiler optimization passes. Optimizations specific to exploiting aspect knowledge, however, 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 the source code weaving approach are:

- Language dependency. Source code weaving is written explicitly for the syntax of the input language.
- Limited expression power. Aspects are limited to the expression power of the source language. For example adding multiple inheritance to a single inheritance language.

Intermediate Language weaving overcome these drawbacks.

Intermediate Language weaving

Weaving aspects through intermediate language gives more control over the executable program than source code weaving. This because combinations of intermediate language expressions, which are not expressible in the source code are possible. Although the IL may be hard to understand, it gives several advantages over source code weaving. These are listed below:

- Programming language independence. Once implemented, all compilers generating the target IL output can be used.
- More expression power. It is possible to create IL constructions that are not possible in the original programming language.
- Source code independence. Can add aspects to programs and libraries without the source code.
- Faster compilation. Only modified objects should be recompiled by the native compiler and finally woven.
- Better separated compilation model. First all compiling is done, hereafter weaving. The native compiler has no knowledge about the aspects that will come.
- Adding aspects at load- or runtime. A special classloader or a runtime environment can decide and do the dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend upon the implementation of runtime environment.

However IL adaption has also some drawbacks. These are partially described as advantage for source code weaving.

- Hard to understand. Specific knowledge about the IL is needed.
- Less debug information available. Exact source locations are not available in Microsoft .NET IL.

- More error-prone. Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect.

Adapting the Virtual Machine

Adapting the Virtual Machine prevents weaving the aspects. This technique has the same advantages as Intermediate Language weaving and can also overcome some of its disadvantages. Thereby aspects can be added avoiding re-compilation, re-deployment, and re-start of the application [25, 26].

Unfortunately, modifying the VM has one major disadvantage. The adapted Virtual Machine involves that every system should be upgraded to a version that can work with aspects.

1.1.7 AOP approaches

As the concept of AOP has been embraced as a useful extension to OOP, different AOP techniques have been developed. As described by [12] these differ primarily in:

the way aspects are specified: every technique uses its own aspect language to describe the concerns.

the composition mechanism provided: each technique only provides composition mechanisms.

the implementation techniques provided: e.g. components can be determined statically or dynamically, the support for verification of compositions.

This section will give a short introduction to AspectJ [13] and Hyperspaces [24], which together with Composition Filters [4] are today's main AOP techniques. A detailed description of Composition Filters will be given in section 1.2.

The AspectJ approach

AspectJ [13] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an openly developed Eclipse project in December 2002.

One of the main goals in the design of AspectJ is to make it a *compatible* extension to Java. With compatible four things are meant:

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 *dynamic crosscutting mechanism*. The other is called *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:

```

1 aspect DynamicCrosscuttingExample {
2
3     Log log = new Log();
4
5     pointcut traceMethods():
6         execution(edu.utwente.trese.*.*(..));
7
8     before() : traceMethods {
9         log.write("Entering " + thisJointPoint.getSignature());
10    }
11
12
13    after() : traceMethods {
14        log.write("Exiting " + thisJointPoint.getSignature());
15    }
16 }

```

Listing 1.3: A example of dynamic crosscutting in AspectJ

The points in the execution of a program where the crosscutting behavior is inserted are called *joinpoints*. A set of joinpoints is called a *pointcut*. In the example above "traceMethods" is 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 joinpoint is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specify where the additional code is inserted. In the example both, before and after advice are declared to run at the joinpoints specified by the "traceMethods" pointcut.

Aspects can contain anything permitted in class declarations as well as definition of pointcuts, advice and some declarations to support static crosscutting. For example, inter-type member declarations which allow a programmer to add fields and methods to certain classes, such as:

```

1 privileged aspect StaticCrosscuttingExample {
2
3     private int Log.trace(String traceMsg) {
4         Log.write(" --- MARK --- " + traceMsg);
5     }
6
7 }

```

Listing 1.4: An example of static crosscutting in AspectJ

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

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

The Hyperspaces approach

The *Hyperspaces* [24] project is developed by H. Ossher and P.Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adapts the principle of multi-dimensional separation of concerns, which involves:

- multiple, arbitrary dimensions of concern
- simultaneous separation along these dimensions
- the ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software lifestyle
- overlapping and interacting concerns (one might think of many concerns as independent or "orthogonal", but they rarely are in practice)

We explain the Hyperspaces approach by an example following the *Hyper/J* [29] syntax. Hyper/J is an implementation of the Hyperspaces approach for the Java language. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses bytecode weaving on binary Java class files and generates new class files to be used for execution.

As a first step, developers create hyperspaces by specifying a set of Java class files that contains the code units that populate the hyperspace. One way to do this is by creating a hyperspace specification:

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

Listing 1.5: Creation of a hyperspace

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 one can specify concern mappings, which describe how existing units in the hyperspace relate to concerns in that dimension:

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

Listing 1.6: A specification of concern mappings

The first line indicates that, by default, all units contained within the package are in 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. One should note that later mappings override the first one.

By means of hypermodule specifications one can define hypermodules, which are modules based on concerns. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

```

1  hypermodule Pacman_Without_Debugging
2      hyperslices: Feature.Kernel, Feature.Logging
3      relationships: mergeByName

```

Listing 1.7: Defining a hypermodule

As this example shows, a hypermodule consist of two parts. The first part specifies the set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. In this hypermodule, the Kernel and Run concerns are related by a "mergeByName" integration relationship. This means that units in the different concerns correspond when they have the same names ("ByName") and that corresponding units are to be combined; for example, all members in similar classes are merged into one class. The hypermodule results in a hyperslice that contains all the classes without the Debugging feature; no debug() methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularization: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. This lowers the entry barrier, greatly facilitates evolution, and opens the door to non-invasive refactoring and re-engineering. This implies that the approach is especially useful for evolution of existing software.

1.2 Composition Filters

1.2.1 The Composition Filters approach

Composition Filters have been developed by M. Aksit and L. Bergmans at the TRESE group, at the Department of Computer Science of the University of Twente, The Netherlands.

The Composition Filters (CF) model is an extension to the Object-oriented model. It is closely related to Aspect Oriented Programming, in the sense that with CF it is possible to model Aspects, but CF dates further back in time. The base concept in CF is that messages that enter and exit an object can be intercepted, and manipulated in various forms, modifying the way in which the object behaves. To do so, in the CF model, a layer called the *interface part* is introduced. The resulting model and its components are shown in Figure 1.2.

The most significant components in the CF model are the *input filters* and *output filters*. Each individual filter specifies a particular manipulation of messages. Various filter types are available for different types of manipulations. The filters together compose the behavior of the object, possibly in terms of other objects. These other objects can be either *internal objects* or *external objects*. Internal objects are encapsulated within the composition filter object whereas external objects remain outside the composition filters object, such as globals or shared objects. The

behavior of the object is a composition of the behavior or its internal and external objects. In

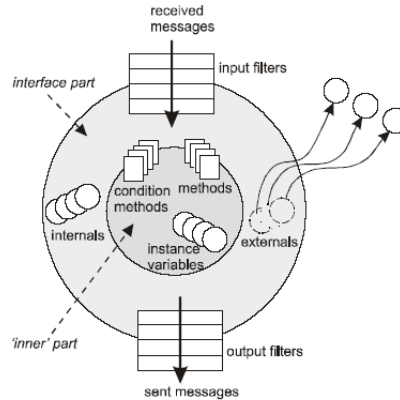


Figure 1.2: The components of the Composition Filters model

addition, -part of- the behavior of the object can be implemented by the 'inner' object, which is therefore also referred to as the *implementation part*. Any conventional object-oriented programming language, such as Java or C# can implement the inner object: the interface part is a modular extension to the inner object.

As mentioned above, there are various filter types, all sharing a common structure; a name that identifies the filter, the type of the filter and a set of expressions that define the way that messages are to be filtered. for each of them the behavior is defined by what actions are taken when it accepts, that is a message matches any of the patterns defined for the filter, or rejects a message. Some common filter types are:

Dispatch: if the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter.

Error: if the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set.

Wait: if the message is accepted, it continues to the next filter in the set. The message is queued as long as the evaluation of the filter expression results in a rejection.

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

Substitute: if the filter accepts, certain properties of the message can be substitute. If the filter rejects, the message will continue to the next filter.

The message interception mechanism of the CF model is explained by means of the example in listing 1.8.

```

1 concern SmartGhost {
2   filtermodule ghostMovement {
3     internals
4     ghost: Ghost;
5     externals
6     pacman: Pacman;

```

```

7   methods
8     int getFleeMove();
9     int getHuntMove();
10  conditions:
11    pacman.isEvil;
12  inputfilters
13    move: Substitute = (
14      isEvil => [ghost.getNextMove] inner.getFleeMove,
15      True => [ghost.getNextMove] inner.getHuntMove );
16    disp: Dispatch = { inner.*, ghost.* }
17  };
18
19  implementation begin in "Java";
20  public class SmartGhostImpl {
21    public int getFleeMove() {
22      // return best flee move
23      return fleeMove;
24    }
25
26    public int getHuntMove() {
27      // return best hunt move
28      return huntMove;
29    }
30  }
31 };

```

Listing 1.8: Example of a filtermodule specification

The example uses a Substitute and a Dispatch filter. The substitute filter will, whenever condition “isEvil” is true and the name of the message is “ghost.getNextMove”, substitute the message with “inner.getFleeMove”. Then, if the message is still “ghost.getNextMove”, it will be substituted with “inner.getHuntMove”. “inner.*” and “ghost.*”. The Dispatch filter accepts all the methods on the interface of class SmartGhost and the class of the internal ghost object: Ghost. The pseudo variable “inner” refers to the implementation of the current instance of SmartGhost.

The superimposition mechanism

In order to add crosscutting concerns to the one or more objects, the composition filters model provides the superimposition mechanism. Superimposition is expressed by a superimposition specification, which specifies how the concerns crosscut each other.

```

1  concern Tracing {
2
3    filtermodule tracingModule {
4      externals
5        log: Log;
6      inputfilters
7        logIn: Meta = ( isEnabled => [ *.* ] log.traceMessage );
8      outputfilters
9        logOut: Meta = ( isEnabled => [ *.* ] log.traceMessage );
10   };
11
12   superimposition {
13     selectors
14       withTracing = { *=Pacman, *=Ghost, *=World };
15     filtermodules
16       withTracing <- tracingModule;
17   };

```

```
18
19  implementation begin in "Java";
20
21  public class Log {
22
23      public void traceMessage(Message m) {
24          // tracing functionality here
25          // ..
26          // continue evaluating this message
27          m.fire();
28      }
29
30  }
31  end;
32
33  };
```

Listing 1.9: Example of a crosscutting concern

The example in listing 1.9 shows a concern that specifies a filtermodule `tracingModule` that filters every incoming and outgoing message, reifies it and passes it to an external of type `Log`, which will log the incoming or outgoing message in a, here unspecified, manner.

The superimposition clause specifies on which instances of classes this filtermodule is superimposed. In this case, the filtermodule `tracingModule` is superimposed on all instances of classes `Pacman`, `Ghost` and `World`.

1.2.2 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 the Compose* project.

- 1985** : The first version of Sina was developed by Mehmet Aksit. This version of Sina contained a preliminary version of the composition filters concept called semantic network. The semantic network construction served as an extension to objects like classes, messages or instances. These objects could be configured to form other objects such as classes from which instances could be created. In this version an object manager took care of synchronization and message processing of an object. The semantic network construction could express key concepts like delegation, reflection and synchronization [22].
- 1987** : Together with Anand Tripathi of the University of Minnesota the Sina language was further developed. The semantic network approach was replaced with declarative specifications and the interface predicate construct was added.
- 1991** : The interface predicates were replaced by the dispatch filter and the wait filter took over the synchronization functions of the object manager. Message reflection and real-time specifications were handled by the meta filter and the real-time filter [23].
- 1995** : The Sina language with Composition filters was implemented using Smalltalk [22]. The implementation supported most of the filter types. Also this year, a preprocessor providing C++ with composition filters support was implemented [15].
- 1999** : The Composition Filters language ComposeJ [32] was developed and implemented. The implementation consisted of a preprocessor capable of translating Composition Filter specifications into the Java language.
- 2001** : ConcernJ [6] implemented as part of a M. Sc thesis. 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.

Chapter 2

Compose*

2.1 Introduction

The .NET platform is gaining more and more acceptance in many different fields of software engineering. There are lots of companies which are largely dependent on the Microsoft tool-set but need or want to use AOP. The Compose* project is addressing these needs with its implementation of the Composition Filters approach on the .NET platform. The Compose* project has two main goals. Firstly, it combines the .NET framework with AOP through Composition Filters. Secondly, Compose* offers superimposition in a language independent manner. The .NET intermediate language supports this. The Composition Filters are declared as an extension of the object-oriented mechanism as offered by .NET. The implementation is therefore not restricted to any specific object-oriented language.

The first section presents an overview of the .NET architecture and highlights the various features of .NET framework. Subsequently it makes a comparison between the .NET Common Language Runtime and the Java Virtual Machine. The features explicit to Compose* are discussed after this. The next section presents the architecture of Compose* and explains all the steps and tools in this architecture.

2.2 Overview of the .NET architecture

The .NET Framework is Microsoft's next step in the evolution of programming [7]. It is a cleanly designed, consistent, and modern API providing support for component-based programs and Internet programming. The main reason Microsoft developed the .NET Framework was the lack of support of the old Windows API for new programming concepts.

This new API has become an integral component of Windows and was designed to fulfill the following objectives [9]:

- To provide a consistent object-oriented programming environment where object code is stored and executed locally, executed locally but Internet-distributed, or executed re-

motely.

- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework consists of two main components [9]: the Common Language Runtime (CLR) and the .NET Framework class library. The CLR is the agent that manages code at execution time, providing the core services. Code that targets the CLR, i.e. code that makes use of the core services, is known as *managed code*. *Unmanaged code*, on the other hand, is code that does not target the CLR (e.g. the executable code is stored in the native machine language). Managed code has to conform to the Common Type Specification, which will be described in more detail in section 2.2.3. 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). Managed code is stored in an intermediate language format, i.e. platform independent, officially known as Common Intermediate Language (CIL) [30]. A detailed description of the CLR is given in section 2.2.1. The .NET Framework class library is a comprehensive collection of object-oriented, reusable types for .NET application developers. In section 2.2.2 a short description of the class library is given.

Figure 2.1 shows the relationships between the Runtime, the class library and an application (managed or unmanaged) in the .NET Framework. The .NET Framework is Microsoft's implementation of the Common Language Infrastructure (CLI) and in this context the CLR is simply called the .NET Runtime or Runtime for short.

2.2.1 Features of the Common Language Runtime

The CLR provides the core services for managed components, like memory management, thread execution, code 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. 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 called just-in-time (JIT) compiling. This process allows the development of CLR's

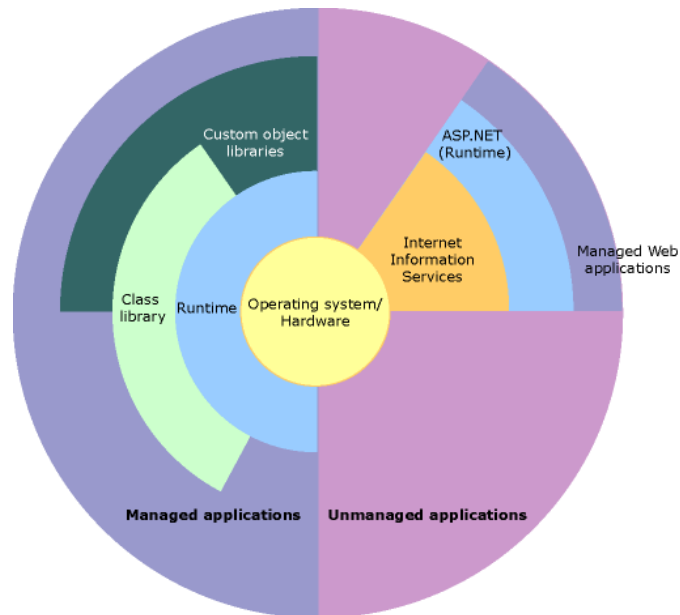


Figure 2.1: The context of the .NET Framework.

Source: *Overview of the .NET Framework* [9].

for any platform, creating a true interoperability infrastructure [30]. The .NET Runtime from Microsoft is actually a specific CLR implementation for the Windows platform.

Microsoft has taken the concept of "any platform" very broad by releasing the *.NET Compact Framework* especially for devices such as personal digital assistants (PDAs) and mobile phones. Because the .NET Compact Framework is a subset of the normal .NET Framework, not only can any .NET developer easily write mobile applications, also easy interoperability between mobile devices and workstations/servers can be implemented [8].

At the time of writing, the .NET framework is the only advanced Common Language Infrastructure (CLI) implementation available. A shared-source¹ implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [28]. Also Ximian is working on an open source implementation of the CLI under the name Mono (<http://www.go-mono.com/>), targeting both Unix/Linux and Windows platforms. Another, somewhat different approach, is called Plataforma .NET (<http://people.ac.upc.es/enric/PFC/Plataforma.NET/p.net.html>) and aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

2.2.2 The .NET Framework class library

The collection of reusable types from Microsoft for the CLR is called the .NET Framework class library. This class library is object oriented and provides integration of third-party components with the classes in the .NET Framework. In this way a developer can use components provided by the .NET Framework, other developers and his own components without worrying about things as version conflicts.

¹Only non-commercial purposes are allowed.

A wide range of common programming tasks (e.g. string management, data collection, 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);
- ASP.NET applications;
- XML Web services;
- Windows services.

2.2.3 Standardization

The entire CLI has been documented, standardized and approved [17] by the European association for standardizing information and communication systems, Ecma International.² Benefits of this standardization for developers and end-users are:

- 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 implementation can communicate with each other, providing applications with easy cross-platform communication means.

Interoperability is, for instance, achieved by using a standardized metadata 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 CLS is a subset of the CTS, 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 [10]. 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 theory³, 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, such as Delphi.NET, Perl.NET, Python.NET and Eiffel#. These programming languages cover a wide-range of different programming paradigms, such as classic imperative, object-oriented, scripting, and declarative lan-

²An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at www.ecma-international.org.

³Unfortunately Microsoft didn't submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.

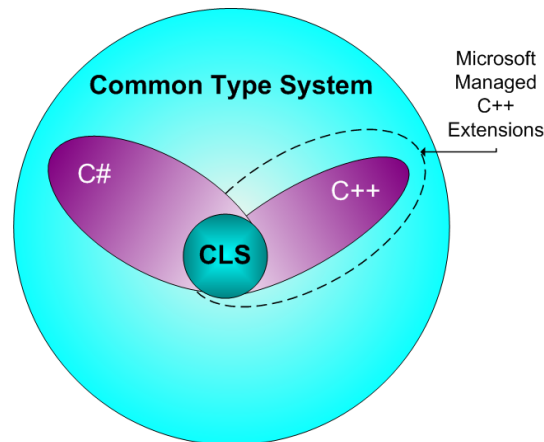


Figure 2.2: The relationships in the CTS.

guages. 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 compiled code is stored and distributed in CIL format, the code can run on any CLR. For cross-language usage the code has to comply with the CLS. Any application can use the class library for common and specialized programming tasks. This class library is also available to the developers. Finally, the integration of the CLR with the platform it is running on is shown.

2.2.4 A comparison between the .NET CLR and the Java VM

Comparisons between Java and .NET have been the starting point for many heated discussions. Still, it is an interesting comparison since these products fight, at least partially, for the same market.

First of all, it is important to recognize the similarities between the products. 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 possibly the lack of an integrated language and platform independent object sharing mechanism in Java. For Java this functionality is provided by this party CORBA implementors while it is tightly integrated in the .NET framework.

To compare the runtime environments we need to recognize the different philosophies behind the two products. While Java's strategy is "One language for all platforms" the .NET philosophy is more like "All languages on one platform". However these philosophies are not as strict as they seem. As noted in 2.2.2 there is no technical obstacle for other platforms to implement the .NET framework and in practice this is already being done. On the other hand, there are also compilers for non-Java languages like Jythong (Python) [18] and WebADA [1] available for the JVM. However, it must be noted that the JVM lacks a language compatibility layer like the CLS. Thus, the JVM in its current state, has difficulties supporting such a vast 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

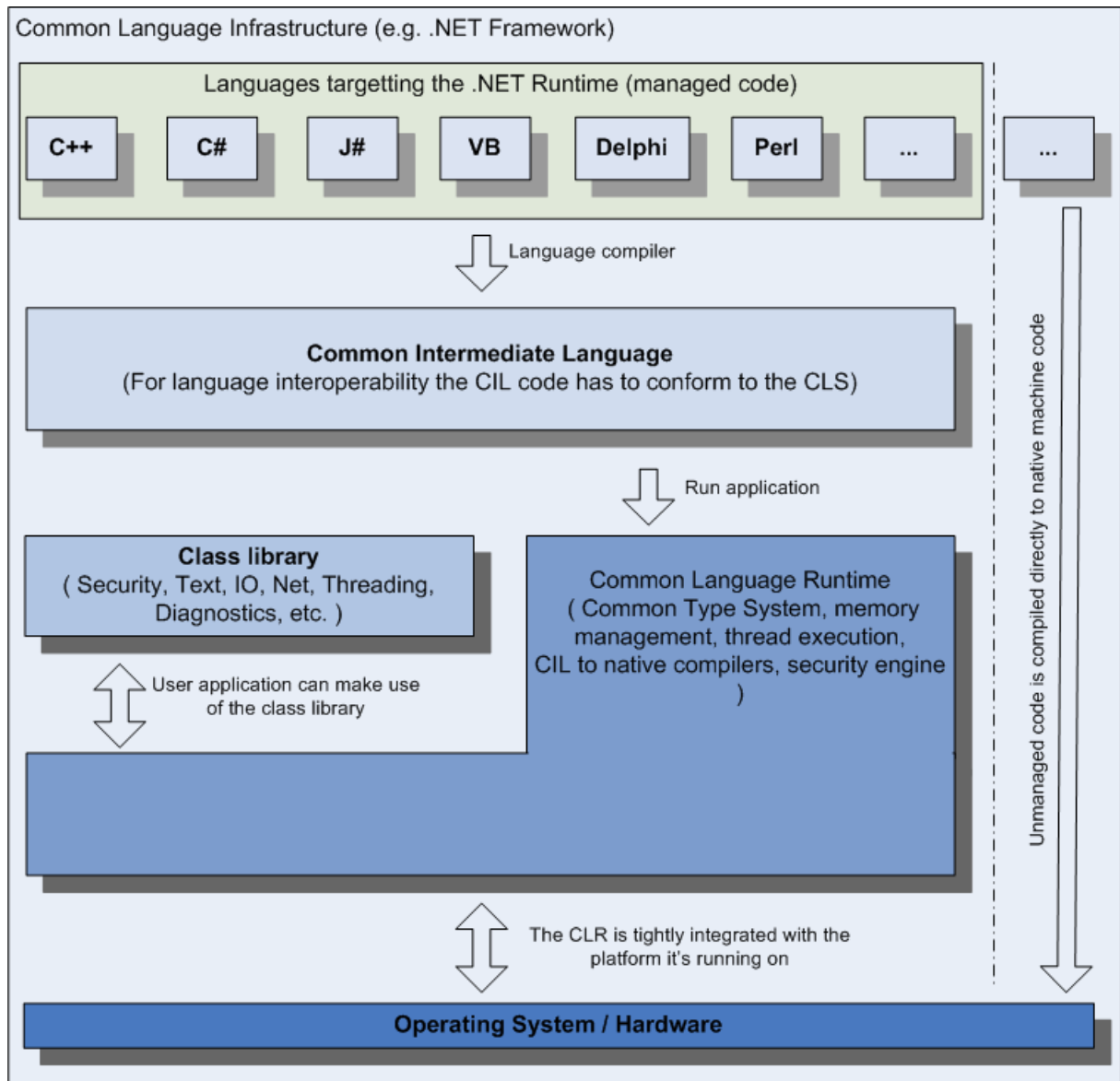


Figure 2.3: The main components of the CLI and their relationships. The right hand side of the figure shows the difference between managed code and unmanaged code.

features, they do so in different ways. While the JVM is a virtual machine interpreting the bytecode, the CLR is a JIT which means that bytecode is compiled into platform specific code just before execution. In theory, this gives the CLR a speed advantage over the JVM. However, many modern JVM's use JIT technology in practice and level out any theoretical advantage the CLR might have.

2.3 Features explicit to Compose*

The Compose* system has four major features which allows for more control and correctness over the application under construction. These features are briefly outlined here.

- One can specify how the superimposition of the filtermodules can or should be ordered. This idea is not new of being able to specify orderings on the superimposition is not new; AspectJ uses the precedence mechanism which uses the “declare precedence” identifier to specify which order is preferred. The implementation we facilitate also provides the possibility for condition execution, depending on the result of execution of filters different execution paths can be achieved. Both mechanisms are specified in the concern definition.
- The ability to detect consistency conflicts is the second feature of Compose*. The Consistency Reasoning Engine(CORE) is able to detect conflicts that may occur when a superimposition has been made and the conjunction and the ordering of filters creates a conflict. As an example imagine a set of filters where the first filter only evaluates method *m* and another filter only evaluates methods *a* and *b* then the last filter is only reached with method *m*; this is consequently rejected and as a result the superimposition may never be executed. There are different scenario's possible that lead to these kinds of problems, e.g. conditions that exclude each other.
- Another major feature of Compose* is its ability to reason about possible semantic problems that may occur when multiple pieces of advice are added to the same joinpoint. The Semantic Reasoning Tool(SECRET) analyzes the filters with respect to their types and possible actions that those filters will do. An example of such a problem is the situation where a real-time filter is followed by a wait filter. Because the wait filter can wait indefinite the real-time property imposed by the real-time filter may be violated.
- The above specified conflict analyzers all work on the assumption that the behavior of every filter is known. Except for the meta filter, the behavior of the filters is well defined. The meta filter can be seen as an around advice in AspectJ, the current message is send as a parameter to an user object. The object can then change or monitor certain aspects of the message or system. This object may decide to return the call or not. These undefined and therefore unpredictable behavior poses a problem to the analysis tools. This feature specifies the behavior of the user object and offers an interface to the analysis tools to incorporate this information.

It should be apparent that the three former features can be implemented in Compose* with relative ease. AspectJ and Hyper/J use the full Java syntax, which is convenient when programming advice. However, it makes reasoning about the same advice difficult, there are and have been a lot of efforts with respect to reasoning about source code. Here the reduced syntax of Composition Filters becomes an advantage, it makes it possible for the tools to do the reasoning they do.

2.4 Demonstrating example

To illustrate the complete Compose* tool-set this section introduces a *Pacman* example. The *Pacman* game is a classic arcade game in which the user, represented by the *pacman*, moves in a maze to eat all the vitamins. Meanwhile the *Pacman* is being chased by *Ghosts*, these *Ghosts* will try to eat the *Pacman*. There are however four mega vitamins in the *Maze* that makes the *Pacman* über. In it's über state the *Pacman* can eat the *Ghosts*.

A simple list of requirements for the *Pacman* game is briefly discussed here:

- If the *Pacman* is being eaten by a *Ghost* the number of lives should be decreased, if no more lives are left the *Pacman* will die.
- Whenever the *Pacman* eats a vitamin or a ghost the score should be updated.
- The *Ghosts* should be able to see if the *Pacman* is über.
- The *Ghosts* should know where the *Pacman* is currently located.
- The *Ghosts* should depending on the state of the *Pacman* try to hunt or flee from the *Pacman*.
- If all the vitamins in the *Maze* are eaten a new level should be started, the difficulty should also be increased.

2.4.1 The object-oriented design

The object-oriented design of the *Pacman* game is presented in figure 2.4.

Each class in diagram 2.4 will be briefly discussed below:

Glyph This is the superclass of everything that moves. A lot of common information is put into this class, for instance the direction and speed. The *Pacman* and *Ghosts* classes can override behavior,

Pacman The *Pacman* class is the representation of the user controlled element in the game. It has some extra functionality like the *Pacman* is über or not.

Ghost This is the representation of the ghosts chasing the *Pacman*. They have an extra property that indicate whether they are scared or not (depending on the über state of the *Pacman*).

Keyboard This class accepts all the keyboard input and makes it available to the *Pacman*.

World The *World* class has all the information about the maze, it knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from the *Glyph* class checks whether movement in the desired direction is possible.

Game The *Game* class encapsulates the control flow of the game and controls the state of the game.

View The *View* class is purely used for painting the maze and the *glyphs*.

Main This is the entry point of the game.

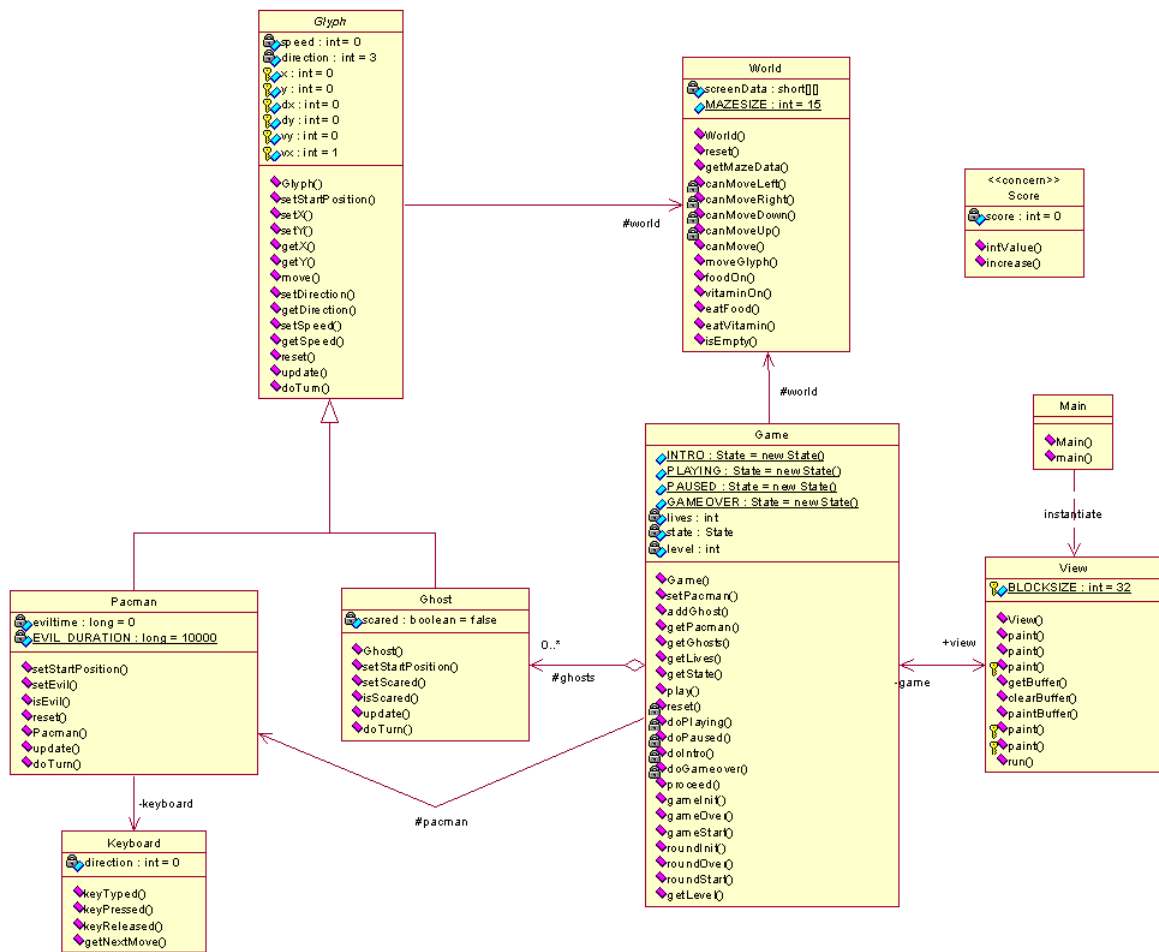


Figure 2.4: The UML class diagram of the object oriented Pacman game.

2.4.2 Completing the Pacman example

The previously described object-oriented design does not implement all the system requirements that were stated. The *Ghosts* should detect if the *Pacman* is evil or über. We create a concern which replaces the original *Ghost*. At every place in the original code where a new instance of the *Ghost* is created, a new *SmartGhost* is created instead. This *SmartGhost* returns a move in the direction of the *Pacman* if the *Pacman* is not evil. Otherwise it returns a direction away from the *Pacman* otherwise. The definition of this concern is given in listing 2.1.

```

1 concern SmartGhost {
2   filtermodule ghostMovement {
3     internals
4     ghost: Ghost;
5     externals
6     pacman: Pacman;
7     methods
8     int getFleeMove();
9     int getHuntMove();
10    conditions:
11    pacman: isEvil;
12    inputfilters
13    move: Substitute = (
14      isEvil => [ghost.getNextMove] inner.getFleeMove,
15      True => [ghost.getNextMove] inner.getHuntMove );
16    disp: Dispatch = { inner.*, ghost.* }
17  };
18
19  implementation begin in "Java";
20  public class SmartGhostImpl {
21    public int getFleeMove() {
22      // return best flee move
23      return fleeMove;
24    }
25
26    public int getHuntMove() {
27      // return best hunt move
28      return huntMove;
29    }
30  }
31 };

```

Listing 2.1: SmartGhost concern in Compose*

The concern uses a *substitute* filter to change the *ghost.getNextMove* call to the method with the correct behavior. After the *substitute* filter it dispatches the call, if it matches, to either *getFleeMove()* or *getHuntMove()*. If it does not match it will dispatch to the internal *Ghost* object, as we want to reuse the basic functionality of the *Ghost*.

The final system requirement that needs to be added to the existing *Pacman* is the score. The score is updated each time the *Pacman* eats something, be it a vitamin, mega vitamin or Ghost. The amount depends on the current level and the state of the *Pacman*. The score is set to zero when a game is initialized. The score is also updated when a level is completed. The score itself has to be painted on the maze canvas to relay it back to the user. These events are all related to the score and are scattered over multiple objects; *Game*, *World*, *View* and *Pacman*. Therefore the score is identified as a crosscutting concern.

The *Score* concern is divided into two filtermodules. The first *scoreModule* is listed in listing 2.2.

```

1  ...
2  filtermodule scoreModule
3  {
4      internals
5          impl: ScoreImpl;
6      externals
7          game: Game;
8      methods
9          eatFood(int, int);
10         eatVitamin(int, int);
11         eatGhost(Ghost);
12         roundOver();
13         gameInit();
14     inputfilters
15         food: Meta = ( [* .eatFood] impl.eatFood );
16         vita: Meta = ( [* .eatVitamin] impl.eatVitamin );
17         kill: Meta = ( [* .eatGhost] impl.eatGhost );
18         round: Meta = ( [* .roundOver] impl.roundOver );
19         reset: Meta = ( [* .gameInit] impl.gameInit );
20 };
21 ...

```

Listing 2.2: scoreModule of the *Score* concern

The second filtermodule of the *Score* concern intercepts the paint method and sends the message as a parameter to paint method in the implementation part. This method issues a *send* command on the message. This means that the paint call is executed as it was, but it returns to the meta filter when the call is done. The *fire* command also lets the message execute, but then control is not returned to the meta filter. When the original paint call returns, the score is painted and the filter is done. The filtermodule is listed in listing 2.3.

```

1  ...
2  filtermodule scoreView
3  {
4      internals
5          impl: ScoreImpl;
6      externals
7          view: View;
8      methods
9          paint(Graphics);
10         paint(Message);
11     outputfilters
12         paint: Meta = ( [* .paint] impl.paint );
13 };
14 ...

```

Listing 2.3: scoreView of the *Score* concern

Both filtermodules are superimposed on the objects in the *Pacman* example. The *scoreModule* is imposed on *Game*, *World* and *Pacman*. The *scoreView* modules is only imposed on *View*. The resulting superimposition specification is listed in listings 2.4.

```

1  ...
2  superimposition
3  {
4      selectors
5          scoring = { *=Game, *=World, *=Pacman };
6          view = { *=View };
7      filtermodules
8          scoring <- scoreModule;
9          view <- scoreView;
10 };
11 ...

```

Listing 2.4: superimposition of the *Score* concern

The two pictures are shown in figure 2.5 show the *Pacman* without and with the concerns.

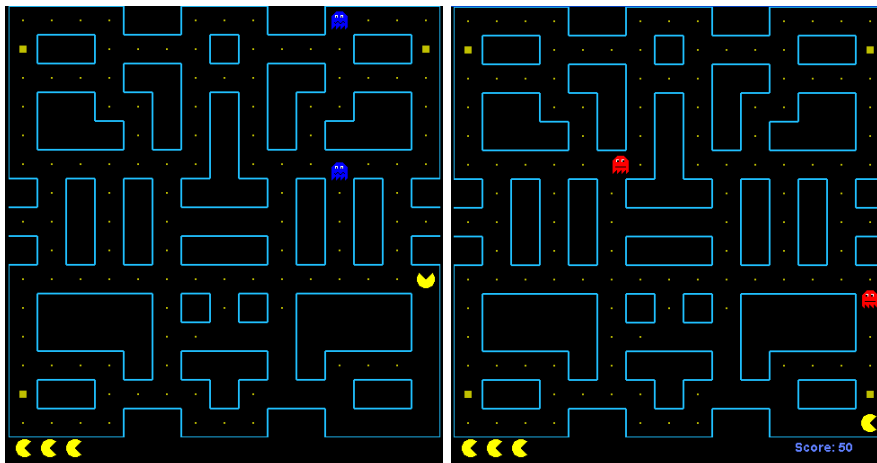


Figure 2.5: Pacman without and with concerns

2.5 Architecture

The entire Compose* tool-set consists of multiple phases and steps. Figure 2.6 shows the entire architecture with current and future work.

All the components shown in the architecture are now explained, the ordering of explanation is control flow of the tool-set.

Master

input: Configuration file (user provided)

description: Master is the initial module to be started when running the Compose* compiler. Master initializes the repository and loads the configuration file. It then proceeds by running the modules in the order presented.

output: If any of the modules run detect an error they through an exception. This exception is caught by master and an error message is presented to the user.

TYM (TYpe Manager pass one)

input: Dummy sources (user provided)

input: Configuration file from Master

description: The sources of the project are extracted from the configuration file. These sources are then compiled with the correct .NET compiler according to the source type. The

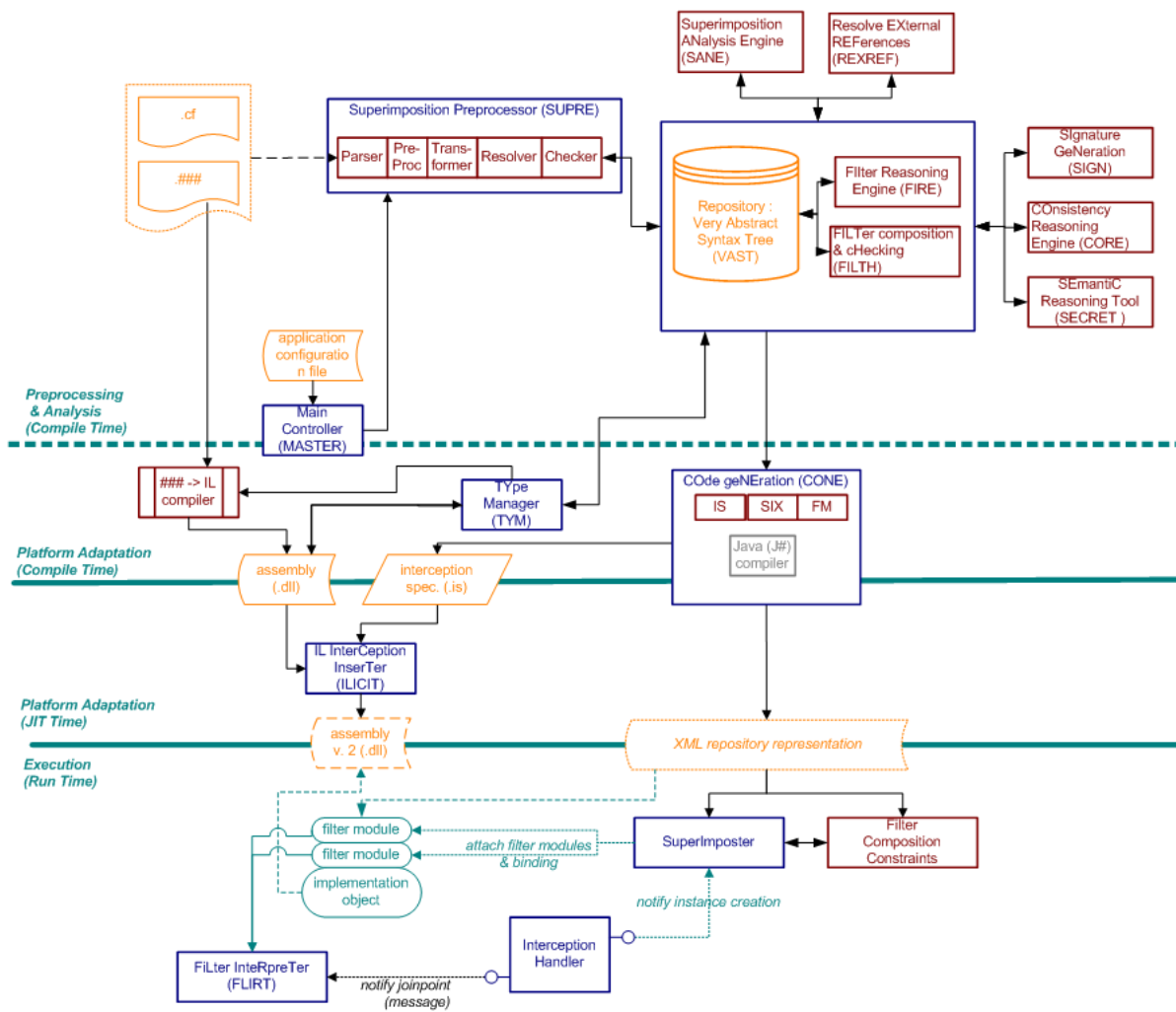


Figure 2.6: The Compose* architecture

resulting assemblies are parsed and the meta-information (type and method signatures) is extracted and put into the repository.

output: Dummy assemblies on disk. Meta-data from dummy assemblies are put into the repository.

SUPRE (SUPERimpositon PREprocessing)

input: Concern sources (user provided) and configuration file from Master

description: SUPRE reads the configuration file and is called once for each concern to be used in the project. Each concern is checked for consistency and an error is thrown if one is found. Each call to SUPRE parses the concern and puts the data into the repository.

output: Concern data put into the repository.

REXREF (RESolve EXternal REFERENCES)

input: Repository data from SUPRE and TYM pass one.

description: Concerns may have both internal and external references, e.g to a method or a condition represented by an object. REXREF traverses the repository and makes sure that all references are resolved.

output: Repository with internal and external references resolved.

SANE (SUPERimposition ANALYSIS Engine)

input: Repository with complete concern and meta-information data.

description: The superimposition analysis engine calculates, for each input specification, the joinpoints where the filtermodules should be imposed. This information is attached to all the imposed objects or concerns.

output: Repository with superimposition resolved.

FILTH (FILTER composition & cHECKing)

input: Repository with super imposition resolved and a filter ordering specification (user defined).

description: SANE produces information about where multiple filtermodules are imposed on the same point. It does not however say anything about the order in which the filtermodules should be applied. The possible orderings are constrained by the filter ordering specification.

output: Repository with filtermodule ordering resolved.

FIRE (FILTER Reasoning Engine)

input: Repository with filtermodule ordering resolved.

description: The Filter Reasoning Engine predicts the result of an incoming messages considering a filter set. FIRE emulates each filter in the filter set and determines possible mappings between the messages and actions. These combinations, with internal states, are stored into the FIRE knowledge base. Providing a convenient interface, FIRE allows other modules querying (and updating) the Reasoning Engine. Modules that use FIRE are CORE, SECRET and SIGN.

SIGN (SIgnature GeNeration)

input: Repository with filtermodule ordering resolved.

description: Composition filters may alter the signature of a concerns. SIGN computes the full signature for all concerns using FIRE and detects if there are filters leading to ambiguous signatures.

output: Repository with the full signature of all concerns.

CORE (COde geneREation)

input: Repository and FIRE.

description: The Consistency Reasoning Engine checks the filter sets, specified by the developer, for inconsistencies. Unreachable filters or actions, conditions with a contradiction or tautology, are examples of problems found by CORE. If a problem is found, the developer will be notified.

output: Warnings about inconsistent filters, actions, or conditions.

SECRET (SEmantiC Reasoning Tool)

input: One concrete order presented by FILTH and an filter specification file (user provided).

description: If multiple filtermodules are imposed on the same joinpoint, certain conflicts may be introduced. The concern containing these filtermodules are often developed at different times and locations by different developers. These filtermodules may have unintended side effects which only effect other filtermodules. If these aspects are combined, semantic conflicts becomes apparent. SECRET aims to reason about these kind of semantic conflicts. It does a static analysis on the semantics of the filters and detects possible conflicts. The used model is, through the use of an XML input specification, completely user adaptable. In input specification, the accept- and reject-actions of filtertypes are specified. Every action is specified by a list of named operations on abstract resources. Also, patterns can be specified that specify the allowed sequence of operations on a resource. When SECRET analyzes a concern, it will fetch the selected filtermodule-order and generate all possible executions of the filterset. Every execution is a unique combination of accept- and reject-actions of the filters in the filterset. The operations of these actions are taken from the input specification and performed on the resources. Then the specified patterns are matches against the sequences of operations performed on the resources.

output: SECRET generates a conflict report which shows where and how the conflicts occur. This report is currently generated as an HTML file.

CONE (COde geNERation)

input: Repository with complete information from all modules.

description: The Code Generator makes all compile time information stored in the repository available at runtime by saving it to a file.

output: The complete repository written to an XML file.

TYM (TYpe Manager pass two)

input: Repository with the full signature of all concerns.

description: The final module called by Master. TYM2 updates the assemblies provided by TYM1 to match the full signatures generated by SIGN. Thereafter the user sources are compiled using the updated dummy assemblies.

output: Compiled user sources.

2.5.1 The Repository

The Repository is the central data-store used by the compile-time part of Compose* (see figure 2.7). Most compile-time modules either rely on data from the repository or compute a result and adds it to the repository. The central repository class is the DataStore which contains a map of objects in stored in the repository. All objects are inserted with a unique key. Objects can be retrieved from the repository either through one of the mass return methods or by requesting an object by its key.

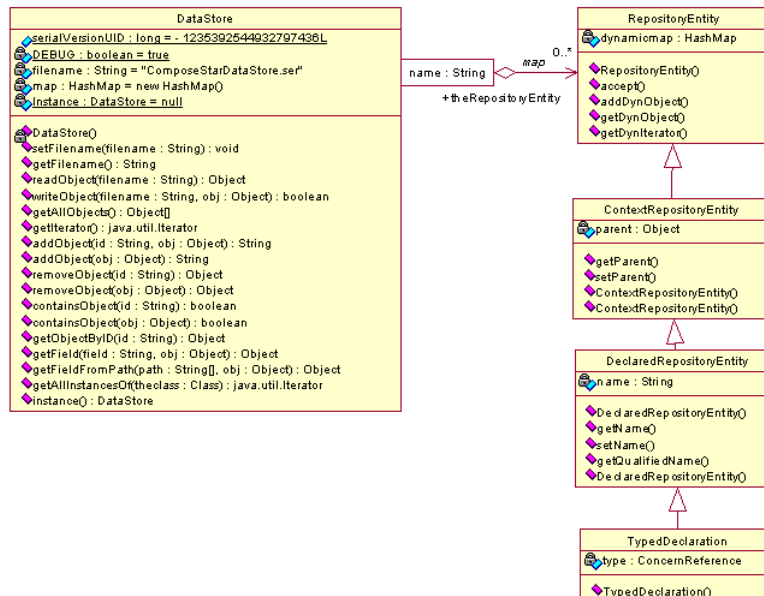


Figure 2.7: UML diagram of the repository.

All objects that are stored in the repository should extend from RepositoryEntity.

RepositoryEntity provides the possibility to add dynamic data to an object. This functionality

can be used to add tool specific data to other repository objects.

The three remaining classes can be extended from depending on the functionality of the class you want to add to the repository:

ContextRepositoryEntity: Some objects in the repository are added as children of other objects. This class adds a pointer to the parent object and makes it possible to step upward in the object tree.

DeclaredRepositoryEntity: All objects containing defining data e.g. a type or method declaration must have a qualified name. This functionality is provided by DeclaredRepositoryEntity.

TypedDeclaration: TypedDeclaration adds a pointer to a Concern, and is therefore the type to inherit from if your data has a concern dependency.

The Repository is the central data-store used by the compile-time part of Compose*. Most compile-time modules either rely on data from the repository or compute a result and adds it to the repository. The central repository class is the DataStore which contains a map of objects in stored in the repository. All objects are inserted with a unique key. Objects can be retrieved from the repository either through one of the mass return methods or by requesting an object by its key.

All objects that are stored in the repository should extend from RepositoryEntity. RepositoryEntity provides the possibility to add dynamic data to an object. This functionality can be used to add tool specific data to other repository objects.

The three remaining classes can be extended from depending on the functionality of the class you want to add to the repository:

ContextRepositoryEntity: Some objects in the repository are added as children of other objects. This class adds a pointer to the parent object and makes it possible to step upward in the object tree.

DeclaredRepositoryEntity: All objects containing defining data e.g. a type or method declaration must have a qualified name. This functionality is provided by DeclaredRepositoryEntity.

TypedDeclaration: TypedDeclaration adds a pointer to a Concern, and is therefore the type to inherit from if your data has a concern dependency.

2.5.2 The Compose* Runtime environment

The Compose* Runtime environment consists of two layers: the JIT Time layer and the Run Time layer (see figure 2.7). In the JIT Time layer ILICIT (IL InterCeption InserTer) inserts additional code (calling the Interception Handler) at the execution joinpoints. These joinpoints are specified in the interception specifications file (XML based) provided by CONE. The modified code is volatile, i.e. it only exist inside the .NET Runtime environment. The Run Time

layer is responsible for really executing the concern code at execution joinpoints. Inside the Run Time layer we can identify the Interception Handler and FLIRT (FiLter InteRpreTer). The Interception Handler is activated at execution joinpoints (recall that ILICIT inserted the necessary calls to the Interception Handler) and will dispatch the necessary calls to FLIRT to enforce the concerns.

ILICIT (InterCeption InserTer)

input: .NET Assemblies from disk and the interception specifications file.

description: To enforce the concern specifications at runtime, execution joinpoints have to be detected. These execution joinpoints are provided by CONE in an interception specification file and are based on the information in the repository. The .NET Intermediate Language (IL) provides a generic way (i.e. for all languages targeting the .NET Runtime) for ILICIT to insert additional code at the execution joinpoints. The task of this inserted code is to notify the Interception Handler, which in turn calls SuperImposter and FLIRT to enforce the concerns.

output: Modified IL code (existing in the .NET Runtime only).

Interception Handler

input: Call from the executing code at execution joinpoints.

description: It is the responsibility of the Interception Handler to accept calls from the executing code and call the necessary methods of SuperImposter and FLIRT and for handling concerns.

output: Calls to SuperImposter and FLIRT.

Filter Composition Constraints

input: XML repository representation and requests from SuperImposter.

description: Filter Composition Constraints provide the runtime alternative of FILTH for SuperImposter. This way SuperImposter can determine the ordering in which different filters should be applied.

output: Answers to the requests made by SuperImposter.

SuperImposter

input: XML repository representation, input from Filter Composition Constraints and calls from the Interception Handler.

description: The task of SuperImposter is to create an internal representation of the concerns, consisting of filters, conditions, internals, externals, etc. for FLIRT. In order to create this internal representation SuperImposter requires a XML representation of the repository and notification of the Interception Handler in case of an instance creation.

output: Calls to create an internal representation of the concerns for FLIRT.

FLIRT (FiLter InteRpreTer)

input: Internal concern representation created by SuperImposter and dispatches from the Interception Handler.

description: The responsibility of FLIRT is to provide runtime execution of Composition Filters. The internal representation of the concerns used by FLIRT is created by SuperImposter. FLIRT will accept messages from the Interception Handler and run them through the internal representation.

output: Execution of Composition Filters.

Chapter 3

Problem Statement

It is always easier for one man to solve another man's problem.

By using meta-filters, CF allows for abstraction of communication among objects into first-class objects called *Advice Types* (ACTs), formerly known as *Abstract Communication Types* [3]. These ACTs allow a programmer to abstract and reuse interaction details. Meta-filters and ACTs are the CF way to create custom advice.

3.1 A Meta-filter example: Jukebox system

To explain the meta-filter we will be using a software-system modeling a jukebox. In our base system, there are three classes. The Jukebox class which models the user-interface of the system. Objects of type Song can be selected (in an unspecified way) and the *play()* method is called on the Song object to play the song. This method will call the *playSong()* method of class Player which is the interface-class of an audio-subsystem.

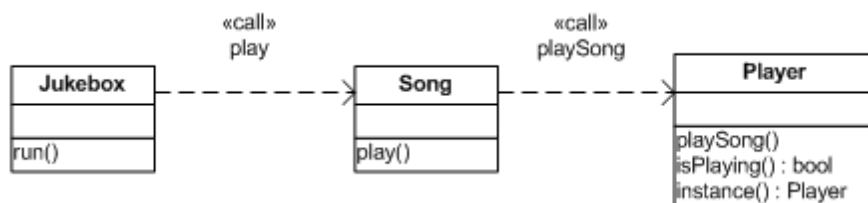


Figure 3.1: UML diagram of the Jukebox base-system

The current design will either block execution until the player has stopped playing or start playing a next song immediately, even when the Jukebox is playing. However, we would like the Jukebox to queue any selected songs if the Jukebox is playing. Listing 3.1 shows a concern that fulfills this requirement.

```

1 concern PlaylistConcern
2 {
3   filtermodule Enqueue
4   {
5     externals
6     playlist: Jukebox.Playlist = Jukebox.Playlist.instance();
7     inputfilters
8     queue: Meta = { [*.play] playlist.enqueue }
9   }
10
11  superimposition
12  {
13    selectors
14    song = { Song | isClassWithName(Song, 'Jukebox.Song') };
15    filtermodules
16    song <- Enqueue;
17  }
18
19 }

```

Listing 3.1: The playlist concern

The superimposition part specifies that the Enqueue filtermodule is super-imposed on the song class. In this filtermodule one input filter is specified: a meta-filter that accepts all *play()* messages. When the meta-filter accepts, the message is reified (a message object is created) and sent as an argument to the *enqueue()* method of the external *playlist* of type Playlist.

```

1 public class Playlist
2 {
3   private Queue songs = new Queue();
4
5   public void enqueue(ReifiedMessage message)
6   {
7     Song song = (Song) message.getTarget();
8     songs.Enqueue(song);
9     message.return();
10    Console.WriteLine("Queued " + song.getName());
11    if( !Player.instance().isPlaying() )
12      play();
13  }
14
15  private void play()
16  {
17    while( songs.Count > 0 )
18    {
19      Song song = (Song) songs.Dequeue();
20      Player.instance().playSong(song);
21    }
22    Console.WriteLine("Queue empty.");
23  }
24
25 }

```

Listing 3.2: Source-code of the Playlist ACT

Class Playlist is listed in listing 3.2. The *enqueue()* adds the Song to a queue. Then *return()* is called on the ReifiedMessage object so no more filters will be evaluated. After this, if the player is idle, *play()* is called which will play Song objects from the queue until the queue is empty.

3.2 The aspect interaction problem

Aspect-orientation extends the object-oriented model, rather than replacing it. One of the extensions provided by the aspect-oriented model is the introduction of a composition operator which we call super-imposition. This operator provides means to *export* behavior to other modules instead of - as in the object-oriented model - having the other modules *import* behavior explicitly.

This exporting characteristic of AOP is very useful with respect to the modularity, adaptability and thus the evolvability of systems and the reusability of modules and aspects. However, understanding the exact behavior of a module, and analyzing or verifying the correctness, requires a deep understanding of all modules and aspects that might affect the behavior of the module.

When multiple aspects are super-imposed on a module, and thus affect the behavior of this module, interferences between these aspect may cause unexpected behavior in a module and might even result in conflicts. To explain this interference we will add another concern to our Jukebox system. Listing 3.3 shows the CreditConcern, which adds functionality that requires the user to pay a credit for every song. One filtermodule, TakeCredits, is super-imposed on the Song class. The filter module has two filters, an error-filter and a meta-filter. The error-filter will only pass the *play()* method if credits have been inserted or raise an exception otherwise. The meta-filter, which is only reached if the error-filter passes, will withdraw one credit and resume the evaluation of the filterset.

```

1 concern CreditConcern
2 {
3   filtermodule TakeCredits
4   {
5     externals
6     credits: Jukebox.Credits = Jukebox.Credits.instance();
7     conditions
8     enoughCredits: credits.payed();
9     inputfilters
10    err: Error = { enoughCredits, true ~> [*.play] };
11    pay: Meta = { [*.play] credits.withdraw };
12  }
13
14  superimposition
15  {
16    selectors
17    song = { Song | isClassWithName(Song, 'Jukebox.Song') };
18    filtermodules
19    song <- TakeCredits;
20  }
21
22 }

```

Listing 3.3: The playlist concern

The diagram in figure 3.2 shows the system extended with both the PlaylistConcern and the CreditConcern. The playlist concern and the credit concern share the same join-point: they both add functionality to Song class and inputfilters of both super-imposed filtermodules match on the *play()* method. When these filters are evaluated, they are in an arbitrary order; either the PlaylistConcern-filters or the CreditConcern-filters are evaluated first. This is shown in figure 3.3.

If the CreditConcern is evaluated first a credit will be withdrawn for a song after which the song

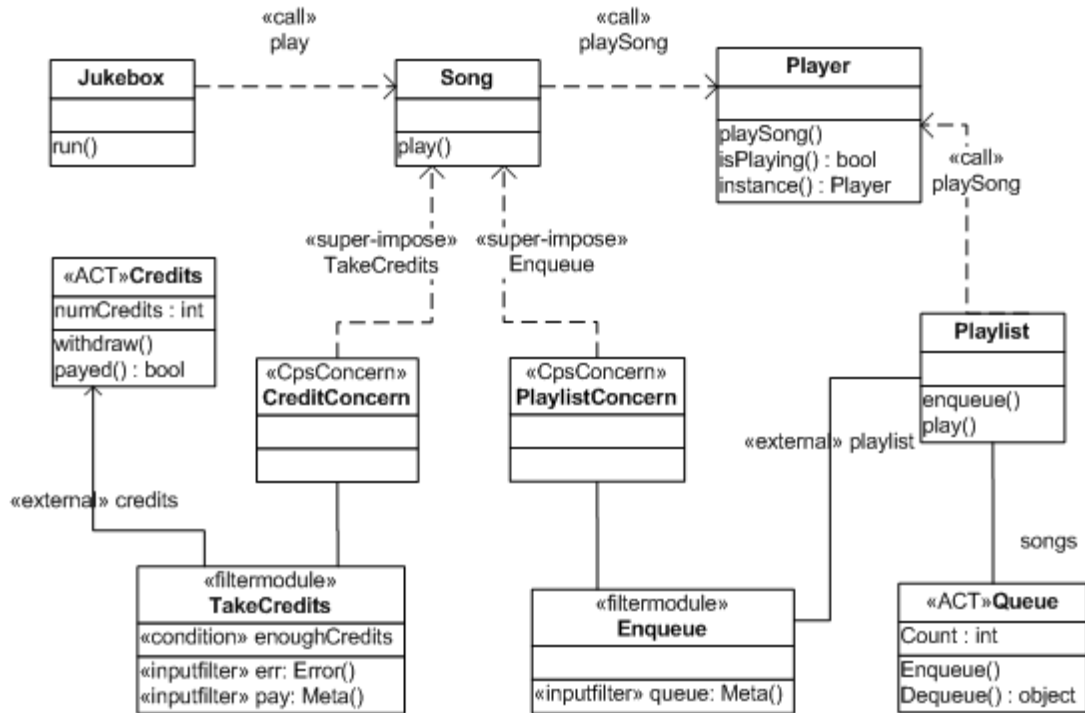


Figure 3.2: UML diagram of the Jukebox system extended with Playlist- and CreditConcern

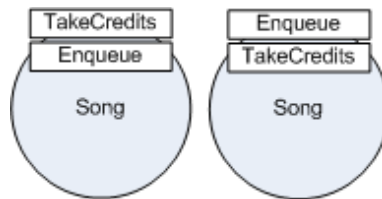


Figure 3.3: Possible filtermodule-orders on class Song

will be added to the playlist. However, when the PlaylistConcern is evaluated first, the song is added to the playlist without any other filters being evaluated. In other words: the song will never be played for.

3.3 Aspect interaction in Compose*

In Compose* the super-imposition mechanism exports filtermodules to concerns. One of the compile-time modules - FILTH - computes all possible orders of the filtermodules that are super-imposed on a concern and selects one of these orders. Some orderings however might be constrained in an ordering specification file. These are filtered out by FILTH.

Compose* is capable of static analysis of only the aspect code - the filtermodules super-imposed on a join-point - since CF is a strongly typed and clearly parameterized aspect language, a requirement that also mentioned by [19]. A semantic description of a filter can be constructed with the knowledge of the filter's type and parameters. Thus, Compose* has become a practical

platform for implementing an engine to detect aspect interference.

3.3.1 Aspect interference and Advice Types

The meta-filter causes severe problems for SECRET, the semantic reasoning engine (see section 2.5). As we have seen in the example in this chapter, a meta-filter will reify a message when it accepts and send this message to an ACT. In these ACTs a developer can not only use regular programming to implement the behavior of the meta-filter, but can also access objects related to the message - such as sender, target and the message arguments - or control the execution of the remaining filters in the filterset.

The meta-filter makes it very hard for SECRET to detect conflicts; on one hand it is not possible to define the semantics of the meta-filter in SECRET's filter specification file, on the other hand the immense capabilities of an ACT make the meta-filter a likely candidate for causing conflicts because of the *turing-complete advice*.

3.4 The Goal

The goal of this thesis is to provide a means for the semantic reasoning engine in Compose* to include meta-filters in its reasoning algorithm and report conflicts introduced by the use of ACTs.

To allow SECRET to include meta-filters in its analysis we will first have to find a way to detect or define the semantics of an ACT-method that is called by a meta-filter. A thorough analysis must be made on the implications of the capabilities of ACTs on SECRET's internal resource-model; new resources or operations might have to be introduced and the order of the operations might be disrupted by an ACT's control over the evaluation of the filterset.

Since SECRET might have to be modified, an analysis will be done of the possibilities to use FIRE, the filter reasoning engine. This tool had not been completed during the development of SECRET and has yet to be integrated. FIRE is able to calculate more precisely what executions of a filterset are possible and allows for querying this information.

Chapter 4

Analysis of Advice Types

It requires a very unusual mind to undertake the analysis of the obvious.

We have seen in the example that the only thing making an ACT a special class is a method that expects an object of type `ReifiedMessage` as argument. The methods of this class can be divided into two categories: manipulation of message parameters and manipulation of the execution of the message. This chapter will explain the operations provided to ACTs by class `ReifiedMessage`.

4.1 Message property manipulation

Message property manipulation operations allow an ACT to control the characteristics of a message. The methods of class `ReifiedMessage` in this category are listed below:

Object `getTarget()`: Returns the current target of a message.

void `setTarget(Object)`: Changes the target of a message to the object given as a parameter.

Object `getSelector()`: Returns the current selector (the name of the called method) of a message.

void `setSelector(String)`: Changes the selector to the passed name.

Object `getArg(int)`: Returns the argument of the message with the given index.

void `setArg(int, Object)`: Replaces the argument with the given index with the passed Object.

Object `getReturnValue()`: Returns the value that will be returned to the caller. This method is only available in *after advice*: the message has been dispatched and is on its way back to the sender. This means that we can use this method after *proceed()*, which is explained later in this chapter.

void `setReturnValue(Object)`: Changes the return-value to the passed Object. This method is, similar to *getReturnValue()*, only available after the message has been dispatched. If one sets the return-value before a dispatch, the return-value is overwritten.

Whether the possibility to modify the target and selector is desirable is questionable. If one uses an ACT for this just because it is conditional, it can probably be replaced with the declaration of an internal or external (for the replacement target), the implementation of a condition-method and the use of a substitute-filter.

4.2 Message execution manipulation

Once a message is passed to the ACT, the ACT can manipulate the execution of the message, e.g. continue the message through the (rest of the) filters in the filterset or stop the execution. The *ReifiedMessage* offers a collection of methods to achieve this. These methods had still to be implemented to be compatible with the threading model of .NET. A part of this thesis was to design and implement these operations. The behavior of these features is presented in this section. A detailed description of the implementation is given in chapter 6.

4.2.1 Resume

First of all we would like an ACT to be able to continue the regular execution of a filterset. In figure 4.1 we see a call coming from the sender. On the *filterset* axis, the activation-blocks of the sequence-diagram represent the action performed when a filter accepts. The first block on the axis represents an accepting meta-filter, where the ACT-method specified in the meta-filter is invoked. During the execution of this message, *resume()* is called upon the reifiedmessage object causing the rest of the filterset to be evaluated. The figure shows this by jumping to the next filter, a dispatch-filter, where the message is dispatched after which control and a return-value is returned to the sender.

resume() does not just cause the continuation of the filterset. After the *resume()* call the ACT will fork (run concurrently) with the execution of the filterset and, after the dispatch, with the sender. The reified message however is "decoupled" from the resumed message: the *getters* can be used but any *setters* will no longer have any influence on the execution of the message, so we cannot do anything harmful or unpredictable by using the reified message.

4.2.2 Proceed

After resuming the reified message, any changes to the message done by other filters or methods will not be visible in the reified message. However, if we would like to see the returnvalue of a message, we would need to be able to read this value from the reified message. To be able to this we use a blocking call to resume the message, which will unblock when the entire execution of the filterset is completed. Any changes to the message are then made visible through the reified message. We call this feature *proceed()*. A diagram of the behavior is shown in figure 4.2. After *proceed()* is called the filterset continues. After the following dispatch, control is returned to the ACT. *resume()* must be used to return the message to the sender. Notice that the behavior is identical to *proceed* as we know it from AspectJ. This implicates that we now have a way to create *around advice*: insert functionality before and after the execution of a method.

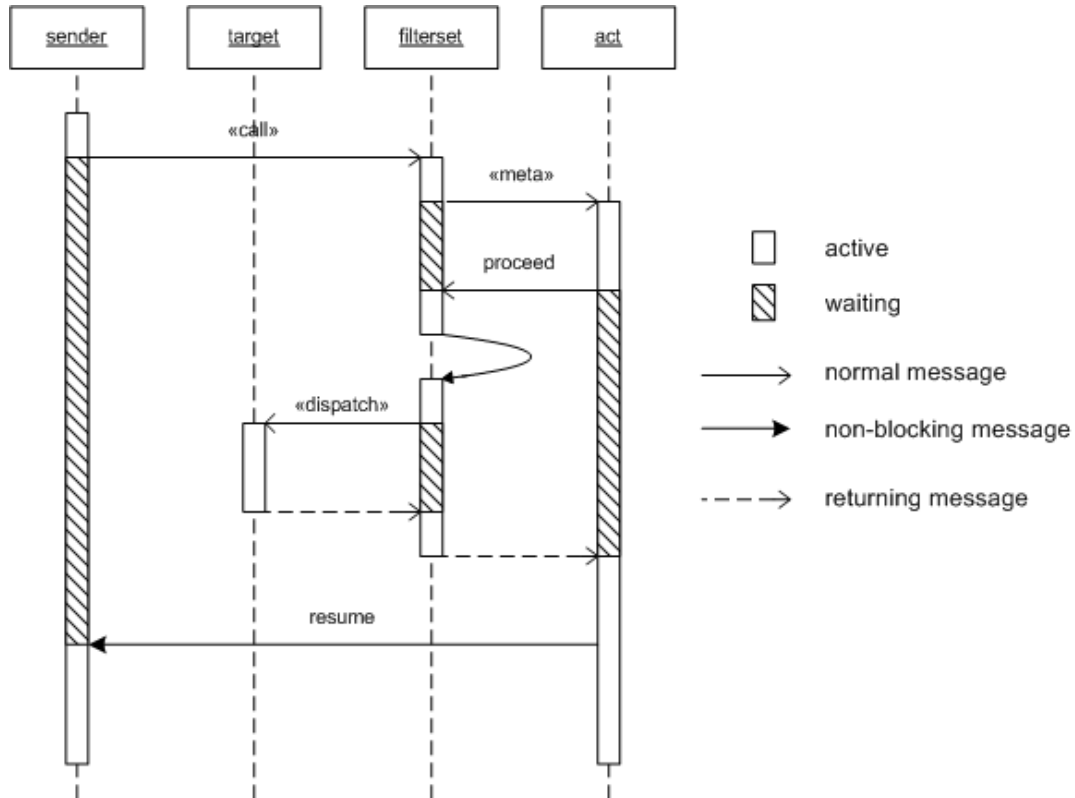


Figure 4.2: An ACT proceeding the message

4.2.5 Send

By calling `send(Object target)` a copy of the message will be sent to the specified target. `Send` has no influence on the execution of the original message. Therefore, this feature will not be discussed in any other chapters of this thesis.

4.2.6 Combining execution operations

Figure 4.5 shows a state-diagram with all the valid combinations of features that can be used during the execution of a single ACT. When the execution of the ACT is finished before the final state is reached the dashed events will be automatically triggered. This diagram also expresses the restrictions as they were already mentioned; `respond()` has to be followed by `proceed()` and `proceed()` has to be followed by `resume()`.

There can be more than one meta-filter in a filterset. All ACTs used by these filters have to restrict to the state-diagram. However, some combinations over multiple ACTs can cause complex behavior that was not expected by a programmer. An important thing to realize is that `proceed()` works *first-in-last-out*. Assume there are two meta-filters and a dispatch-filter and both the ACTs call `proceed()`. After the dispatch, the second `proceed()` will unblock. When this second ACT has finished execution or `resume()` is called by the ACT explicitly, the first `proceed()` will unblock. This means that part of the behavior of the filters is not executed in the

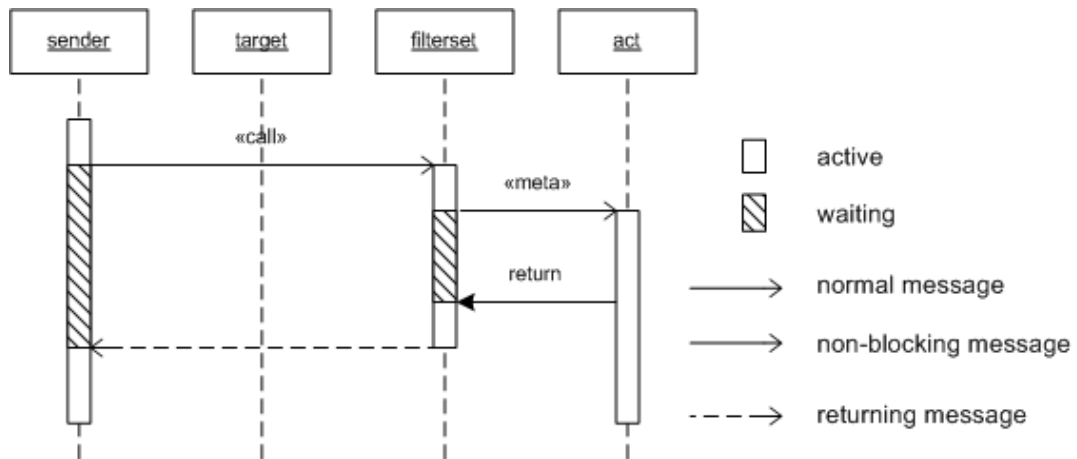


Figure 4.3: An ACT returning the message

same order as the filters are in. Any code after a *proceed()* in the first ACT will be executed last. Another thing to keep in mind is that *respond()* can only be used once for every message. Every message has only one sender and this sender can only be continued once. Another *respond()* call will not do anything.

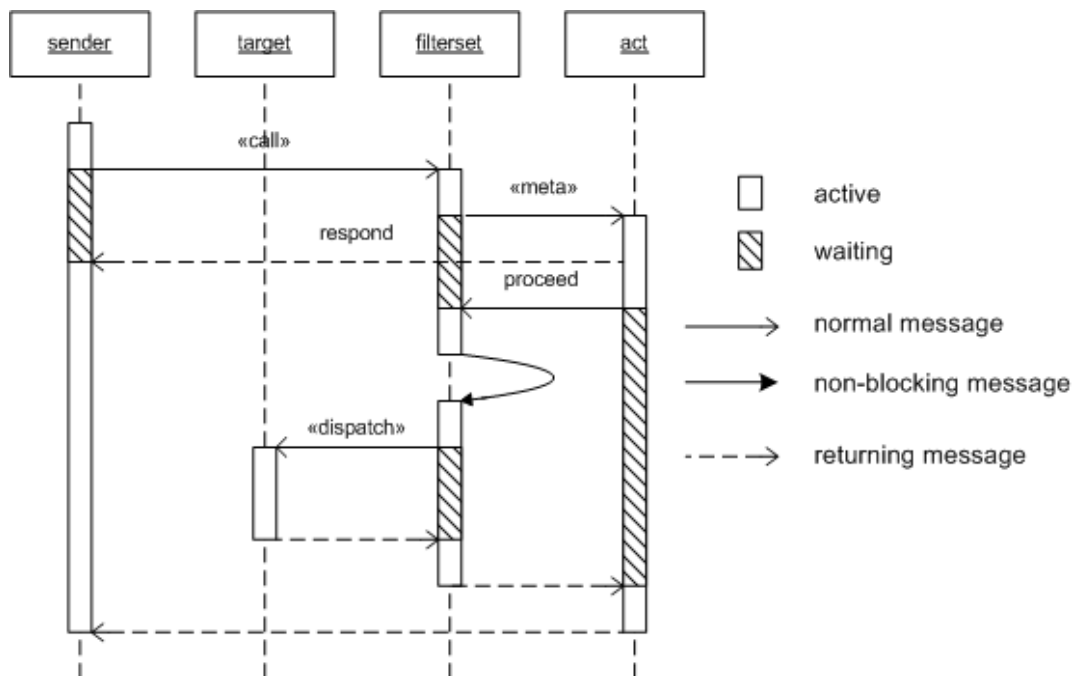


Figure 4.4: An ACT responding the message

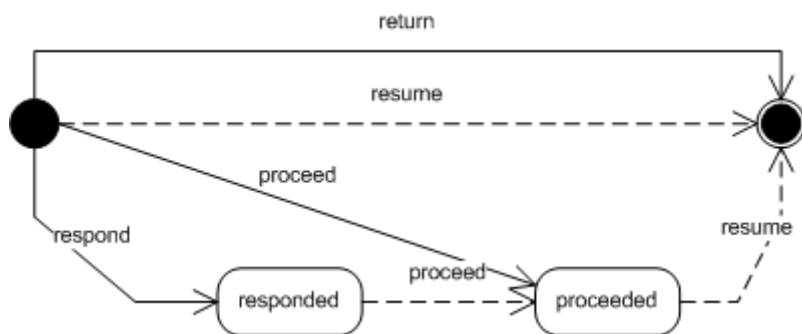


Figure 4.5: Allowed combinations of message execution manipulation

Chapter 5

Solution model

The important thing about a problem is not its solution, but the strength we gain in finding the solution.

This chapter will first describe how the semantics of an ACT can be specified. Then we explain how functionality of an ACT can be described in the resource-model used by SECRET. An operation-ordering algorithm is explained which is required because of a feature of ACTs. Finally, ACT-specific conflicts are explained and detection patterns for these conflict are presented.

5.1 Defining ACT semantics

In the semantic reasoning engine, the semantics of a filter are defined by the type of the filter. For the meta-filter however, the semantics are defined by the method that is called by the meta-filter. To be able to use these semantics in the reasoning process, we need a way to define these semantics. Two possibilities arise: source-code analysis of the ACT and a manual specification of the semantics of an ACT in some language, created by the programmer of the ACT.

Unfortunately, source-code analysis to identifying the behavior of regular (turing-complete) code is not possible. As we will show later in this chapter part of the methods used by the ACT can be translated directly to certain operations on resources, like *setTarget()* and *getTarget()* of the ReifiedMessage object. However, operations on conceptual resources like timing or state would be very hard to identify in the source-code.

The obvious way is to describe the behavior in terms of operations on resources as this maps directly onto the current model of the reasoning engine. However, this means the programmer needs to define the behavior of the ACT twice: once in a programming language and once in the specification language. One could image that this would cause problems when for instance the programmer forgets to add a semantic definition or adapt a semantic definition according to a modification of the ACT's implementation. To keep the chance of these kind of problems

as low as possible we have chosen to have the semantic definition embedded in the source code, directly coupled to the corresponding ACT-method, as opposed to a semantic definition in a separate configuration file.

The semantic definition is specified as a sequence of resource-operations. The operations are separated by a comma. An example of such a specification is "*target.read, selector.write, message.return*". This sequence specifies a *read* operation on the *target* resource, a *write* operation on the *selector* resource and a *return* operation on the *message* resource.

Complex program statements, like an *if-statement*, can not be expressed by a sequence. Also, only one specification per method is allowed. Therefore, a programmer should specify the worst-code scenario of the method, or combine all possible scenarios in one specification.

The means to embed the semantics in the source-code is a design-issue and thus presented in chapter 6.

5.2 Resource-model instantiation

In this section the functionality of class ReifiedMessage presented in chapter 4 is mapped onto the resource-model; for every method the used *resources* and the *operations* on these resources are specified. To be able to give the best result in detecting conflicts, the resource-usage of the dispatch filter and the error filter as specified in [11] have also been extended with some additional operations.

5.2.1 Meta filter

The resource-usage of the matching part of a meta filter is identical to other filters. When the filter reads a condition a *read* operation is performed on the *state* resource. Depending on the specification of the message pattern, *read* operations will be performed on the *target* and *selector* resources.

The substitution part specifies which ACT will be used and what method of the ACT will be executed. However, this information is not used to modify either the target or the selector of the message. When the ACT has finished its execution, and no changes have been made by the ACT to the target or the selector, the target and selector will have the same values as when the message entered the meta-filter. Therefore, no operations will be performed to specify the substitution part.

To specify the behavior of the ACT itself we will give a listing of all the methods of class ReifiedMessage and explain the resource-usage that specifies the behavior.

<code>getTarget()</code>	<code>"target.read"</code>
<code>setTarget()</code>	<code>"target.write"</code>
<code>setSelector()</code>	<code>"selector.write"</code>
<code>getSelector()</code>	<code>"selector.read"</code>
<code>getArg()</code>	<code>"args.read"</code>
<code>setArg()</code>	<code>"args.write"</code>
<code>getReturnValue()</code>	<code>"returnvalue.read"</code>
<code>setReturnValue()</code>	<code>"returnvalue.write"</code>

The operations specified above are all very straight-forward. We will now explain the resource-usage of the methods to manipulate message-execution.

resume() : *Empty specification*

This method actually does not have any behavior that can be specified by resource-operations. The method has a side-effect however. After calling *resume()*, the ACT can no longer affect the execution of the message. This means that any code after *resume()* should not be specified. Operations specified after a return will be discarded.

return() : *"returnvalue.write, target.dispose, selector.dispose, args.dispose, message.return"*

When an argument is passed to this method *setReturnValue()* is called, so we add a *write* operation on the *returnvalue* resource. Then the message is continued but will not be evaluated by any other filters. Instead, the message will return to the sender. To be able to specify this behavior we introduce a new resource *message* and a new operation *return* on this resource.

Since the message is returning to the sender, writing the target, selector or argument afterwards can be pointless. To be able to reason about this we will perform a *dispose* operation on these resources. This operation means that the resource is no longer in use. Identical to *resume()* any code after *return()* should not be specified since the ACT can no longer affect the execution of the message.

respond() : *"message.respond"*

Since this method only triggers the caller to continue execution its behavior does not affect any message properties or the execution of the filterset. We still want to know about the *respond()* call in our reasoning process; to be able to detect possible side-effects, e.g. when a message is responded twice by two different ACTs. Therefore we introduce a *respond* operation on the *message* resource.

proceed() : *"message.proceed"*

This will continue the execution of the filterset until the method has been dispatched and then return control to the ACT. This means that operations that specify the part of the ACT after the *proceed()* call are performed after the operations performed by the dispatch. This is solved by a resorting-algorithm which requires to know when *proceed()* is called. We introduce the *proceed* operation on the *message* resource to identify the resources that require moving.

Conceptual resources Some behavior of an ACT can not be mapped to operations on the standard set of resources. Instead the programmer could introduce a resource that captures

a problem best and use custom operations on it. In the example presented in chapter 3 the code used in the ACTs can not be represented by any of the operations above. We introduce a *song* resource and perform the operations *credit* and *play* on this resource. In other words: a programmer should not always look at the code but sometimes try to specify the *purpose* of a meta-filter or ACT.

Since we have introduced new resources and operations we should specify the usage of these resources for other filter-types as well.

5.2.2 Dispatch filter

A dispatch-filter executes a method but we do not know the semantics of this method. Instead, we can specify a general method with our standard set of resources in the dispatch-filter. This specification is explained below.

The substitution part of the filter can change the target and selector. We start the specification of the dispatch-filter with a *write* operation on the *target* and *selector* resource. After the method is invoked it might use the arguments passed to the method, so a *read* operation is performed on the *args* resource. Concluding the execution, a returnvalue can be returned which is specified by a *write* operation on the *returnvalue* resource. Then a *dispose* operation is performed on *target*, *selector* and *args*. Finally, the message is returned to the sender, which we specify with a *return* operation on the *message* resource. Notice the similarity with a *return()* call by an ACT.

The complete specification of a dispatch-action is "*target.write, selector.write, args.read, returnvalue.write, target.dispose, selector.dispose, args.dispose, message.return*".

5.2.3 Error filter

When an error-filter rejects an exception will be thrown which implicates the end of the filterset. However, since we can *proceed()* a message now, an ACT might expect a return-value instead of an exception. This requires us to specify the exception in the resource-model. An *error* operation is introduced on the *message* resource. We also *lock* the *returnvalue* since it does not have a value. We will use this lock to detect illegal read and write operations. Finally, we add a *return* operation on the message resource, because no other filters will be evaluated.

This results in "*message.error, returnvalue.lock, message.return*".

5.3 Modifying the analysis engine

When the reasoning engine analyzes a specific order of the filtermodules that have been superimposed on a concern, all possible executions - combinations of accept and reject actions of the filters in the filterset - are generated. Then the operations that specify the semantics of those actions are retrieved from an internal data store. These operations are the combination of the operations performed by the matching part and the operations specified for either the accept- or

the reject-action of a filter. A dispatch- or error- action ends the filterset so after these actions no more actions are processed.

As mentioned in this chapter a *proceed* operation on the *message* resource requires resorting of the performed operations. Furthermore, an execution might consist of a meta-action performing a *return* operation on the *message* resource followed later by a dispatch- or error-action. In this case, the dispatch- or error- action will not actually happen so the operations performed by these actions have to be discarded. An algorithm to fulfill these requirements is now explained.

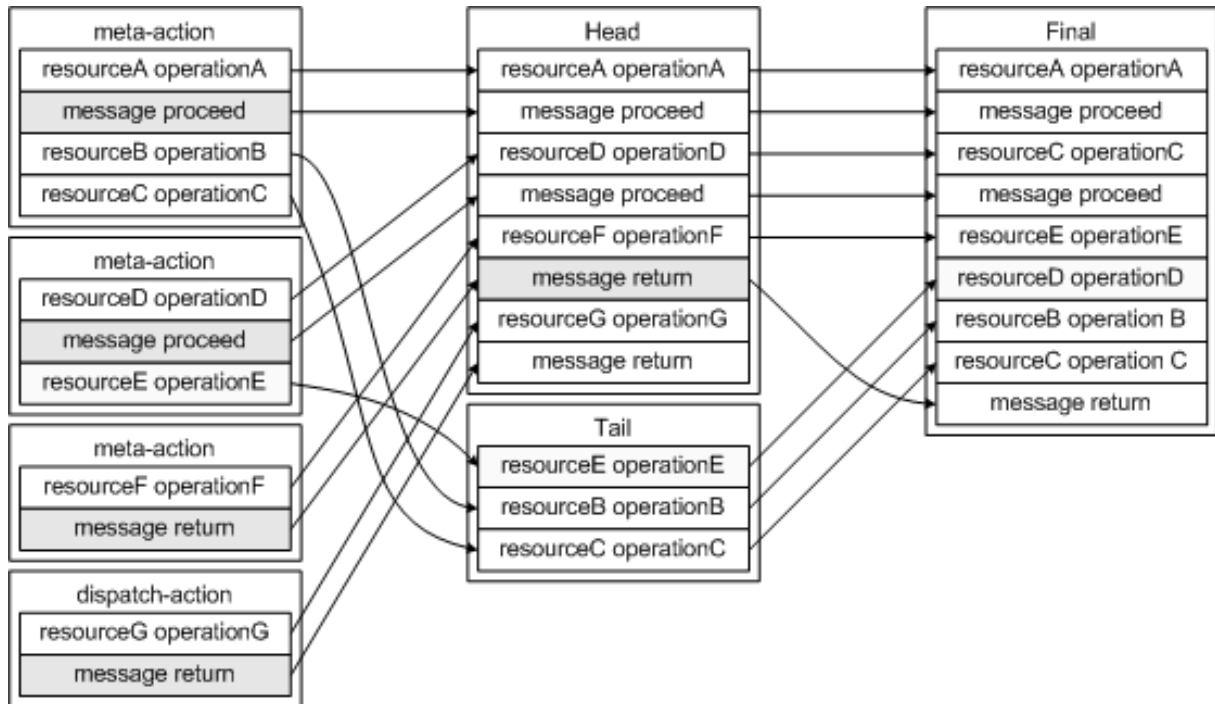


Figure 5.1: Diagram of the operation sorting process.

Figure 5.1 shows the algorithm performed on a list of actions. In the left column four actions are listed. The first three are meta-actions of which the first two perform a *proceed* and the third one performs a *return* operation on the *message* resource. Remember that after *return()* no more behavior should be specified so the *return* should be the last operation of the action. The last block represents a dispatch-action, also ending with a *return* operation on the message resource. Not all operations performed by a dispatch-action have been added since the figure only serves as an example.

The first phase of the algorithm is represented by the first and second column. One action is processed at a time. Operations are added to the *Head* list shown at the top of the second column. Whenever a *proceed* operation is encountered while going through the operations of a meta-action, all remaining operations in the meta-actions are pushed on top of a second list called *Tail*, shown at the bottom of the second column. Notice that in the resulting *Tail* list *operationB* and *operationC* still are in the same order but *operationE* has come before these.

During the second phase - the second and the third column in the picture - the highest *return* operation in the *Head* list is selected. The operation has been marked gray in the figure. All operations after the selected one are removed because these are performed by actions that are

not executed. Also, operations of an action that have been specified after a *message.return* are discarded. Finally, the operations from the *Tail* list are appended to the *Head* list resulting in the list shown in the third column.

5.4 Conflict detection

ACTs give the meta-filter more expressive power than other filters. They therefore can cause conflicts that other filter can not. Using the set of operations introduced in the second section of this chapter we can specify the patterns to detect these conflicts. To simplify the specification of patterns we allow both conflict- and requirement-patterns. Conflict-patterns will cause a warning when the pattern matches the operations on a resource. Requirement-patterns will result in a warning when the operations do *not* match the pattern. A pattern has to match just a part of the entire sequence. To match an entire sequence of the beginning of the end of a sequence we can use the \wedge and $\$$ symbol, which match the begin and the end of a sequence, respectively.

5.4.1 Respond twice

When a message is responded, the sender of the message continues its execution. Another ACT might however also respond the message. The action will not do anything. We want to warn the programmer that the sender has already continued. We can do this by laying a constraint on the *message* resource with conflict-pattern $(respond)(.*)(respond)$. This pattern matches two *respond* operations that can optionally be separated by other operations.

5.4.2 Proceed and error

An ACT that calls *proceed()* followed by an error-filter raising an exception might cause all kinds of unexpected problems. For instance, after the *proceed()* the ACT might try to modify the return-value. Because an error-filter does not set a return-value, this will cause an exception. We present three different patterns to detect these conflict. All patterns can be used by SECRET. Some of them do not always match. However, they give a more detailed message of the conflict.

The first method is to specify a conflict-pattern $(proceed)(.*)(error)$ on the *message* resource. With this constraint we are able to detect if a *proceed* is followed by an *error* operation. There can optionally be other operations in between.

Another way to detect this problem is to use a conflict-pattern $\wedge(read)$ on the *return-value* resource, giving a warning when a *read* is the first operation on the *returnvalue* resource. If an ACT reads the return-value it should either be written by a dispatch-action or by another ACT using *reply()* already. A warning will also be given when *getReturnValue()* is used in an ACT before a message is proceeded.

For the last method we will use the conflict-pattern $(lock)(.*)$ on the *returnvalue* resource, which will match a *lock* operation followed by any other operation. When a resource is locked, no other operations are allowed.

5.4.3 Writing after returning

When an ACT *proceeds* a message, control will return to the ACT after the message is returned by either a dispatch or by another ACT calling *return()* on the reified message. The ACT can then write the target, selector and arguments of the message but these resources are probably not read anymore. We present two ways to detect this problem.

The first way is to specify a conflict on all resources stating that a write should always be followed by at least one read. We can do this with the conflict-pattern $(write)\$, which matches any write operation that is the last operation on a resource.$

The second way to find these problems is to use the *dispose* operation that we have introduced. The conflict-pattern $(dispose)(.*)(write)$, matches a *dispose* followed by a *write* and optionally other operations in between.

Both ways will work. However, the first one is more general and might also detect conflict that we have not anticipated.

5.4.4 Playing a song without paying it

With the patterns specified above we can still not detect the problem in our example; a song is played by the jukebox but no credits are withdrawn. Therefore we have specified the semantics of the ACT with operations on a conceptual resource, *song*. To detect the conflict we have to put domain-knowledge in a constraint. Whenever a song is credited, it should also be played and vice versa. We have put this knowledge in the requirement-pattern $\hat{((credit)(play)) | ((play)(credit)) }?\$, on the *song* resource. This matches exactly three different sequences: $(song)(play)$, $(play)(song)$ and an empty sequence.$

Chapter 6

Design and Implementation

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

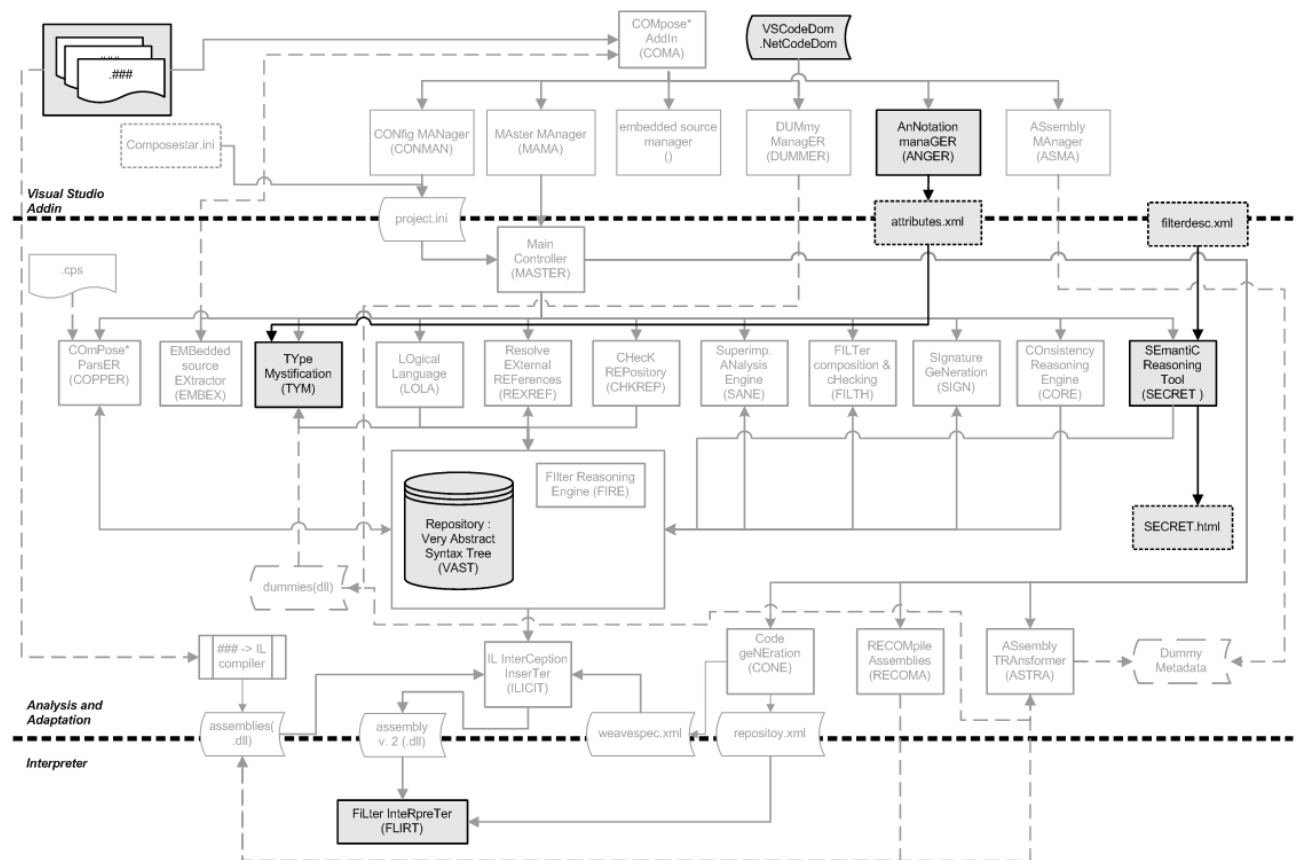


Figure 6.1: Architecture with SECRET highlighted

This chapter describes the effort taken to implement the changes to SECRET to be able to reason about the meta-filter. The implementation of the presented solution is spread over the entire architecture. This is shown in figure 6.1 where the highlighted boxes are in part of the solution.

The architecture consists of three layers. The top layer is a Visual Studio plug-in. The plug-in is mainly used to gather project configuration options and details about the source-files. The first two sections describe how method semantics are added to ACT source-code and how this information is gathered and exported to be used by the second layer, which is executed by the plug-in.

The second layer is the *Analysis and Adaption* layer. This is where SECRET resides. First we will describe how all nessecary information from the input-files - *filterdesc.xml* and *attributes.xml* - is gathered and stored. Then an overview is presented of the analysis process and the report-file (*SECRET.html*) created by SECRET will be shown and discussed.

The third layer, the *Interpreter*, is used when running programs compiled with Compose*. The last section in this chapter presents the implementation of the Meta-filter in the *FiLter InteRpreTer* (FLIRT).

6.1 Specification of semantics using annotations

As explained in chapter 5 we want the semantic specification - a sequence of operations - embedded in the source-code. The adopted way is by using annotations, in .NET known as custom attributes. Annotations are static objects that can be attached to all kinds of code elements such as classes, methods, member-variables and method-parameters. With these annotations we are able to add additional information - metadata - to certain methods.

Adding an annotation containing the semantic definition to all methods called by a meta-filter provides a tight coupling between the code itself and the semantic description. Listing 6.1 shows the *enqueue()* method of the Playlist ACT and the *pay()* method of the Credits ACT from the example shown in the previous chapters.

```

1 public class Playlist
2 {
3     [Semantics("target.read, args.dispose, target.dispose ,
4     selector.dispose , song.play, message.return")]
5     public void enqueue(ReifiedMessage message)
6     {
7         // target.read
8         Song song = (Song) message.getTarget();
9         songs.Enqueue(song);
10        // args.dispose , target.dispose , selector.dispose , message.return
11        message.return();
12        // song.play
13        if( !Player.instance().isPlaying() )
14            play();
15    }
16 }
17
18 public class Credits
19 {
20     [Semantics("song.credit")]

```

```

21     public void withdraw(ReifiedMessage message)
22     {
23         // song.credit
24         this.numCredits--;
25         Console.WriteLine("Payed, " + numCredits + " left");
26         message.resume();
27     }
28 }

```

The methods are annotated with a *Semantics* attribute. The attribute takes a string as argument containing a comma-separated list of resource-operations, where the name of the resource and the operation are separated by a dot. To explain which statements cause the operations, they are also added in the method, as documentation above the line they relate to. Notice in the playlist example that the *play* operation on *song* is added before the *return* operation on *message*. This is because we cannot use any operations after the *return* operation. It does not matter however when the *play* operation is not exactly in the same position in the code because in this case it only matters *that* the song is played and not *when*.

Instead of a string with a sequence of operations, we could have also declared resources first-class, and the allowed operations as public static members of these classes. The annotation would then get an array of these operations as parameter. The advantage of this is that the alphabet of allowed operations on resources is checked by the C# compiler. However, the arguments passed to annotations are collected before the C# compiler is invoked, as will be shown in the next section. We therefore chose to use strings instead. Strings are best readable by the programmer and easier to parse.

6.2 Collecting and storing annotations

From the plug-in, a model is available of the source code contained by the Visual Studio project. This model also includes any annotations used in the source-code. To be able to use the information contained by the annotations in the compile-time part of the Compose* system, these annotations are collected from the code-model and placed in an xml-file called "attributes.xml". Unfortunately the values passed to an annotation are only available when the annotations are used in C#. This is a .NET bug, which might be fixed in future releases. Listing 6.2 shows an example of what this file contains when it is generated for the example of the Jukebox system.

```

1 <?xml version="1.0"?>
2 <Attributes>
3   <Attribute
4     type="Composestar.Reasoning.Semantics"
5     value=""song.credit";"
6     target="Method" location="Jukebox.Credits.withdraw" />
7   <Attribute
8     type="Composestar.Reasoning.Semantics"
9     value=""target.read,message.return,song.play";"
10    target="Method" location="Jukebox.Playlist.enqueue" />
11 </Attributes>

```

The file contains an *Attribute* element for every annotation found in the source-code, all containing the annotation's type, value, target and location.

type: The type of the annotation. When dealing with ACT semantics this will always be

Composestar.Reasoning.Semantics.

value: This is a string containing all arguments passed to the constructor of the annotation. In our case the constructor takes one argument: a string containing a comma-separated list of resource-operations. Visual Studio's code model takes this string from the source-code. The sources are not compiled yet at this time.

target: The target is the type of code-element the annotation is attached to. This target can be set to Type (classes and interfaces), Method, Field (class members) and Parameter. For a Semantics annotation the target is of type Method.

location: The location will be a fully qualified name of the target. In our case this is the method that is executed in the ACT.

6.3 Initialization of the Reasoning Engine

In the compile the XML-file containing the annotations is read and the annotations are stored in the repository. This repository also contains a model of all source-code. The UML diagram in figure 6.2 shows the part of this model to store these annotations.

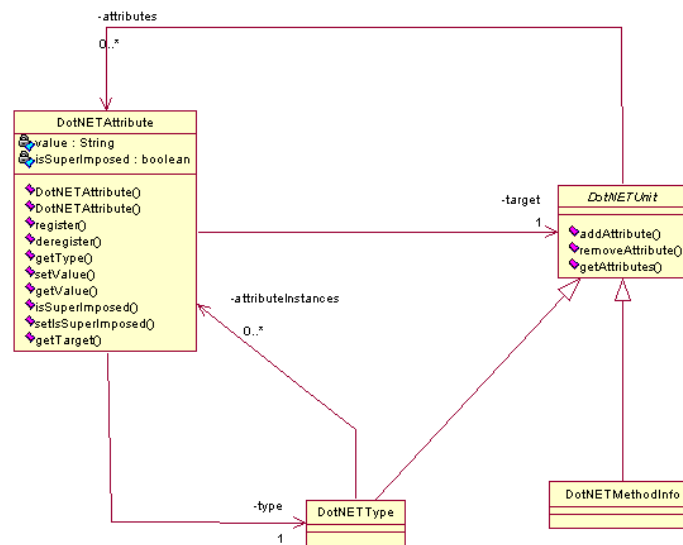


Figure 6.2: Part of the compile-time repository meant for storing annotations

Annotations are represented by objects of class DotNetAttribute. The figure shows a DotNETAttribute class which has a certain type. For a Semantics annotation this type will be the Composestar.Reasoning.Semantics type. The target is of type DotNETUnit. This is the superclass of all code elements in the code-model. Because the reasoning tool only deals with annotations attached to methods we have also displayed the DotNETMethodInfo class to illustrate how annotations are connected to objects of this type.

To find annotations in the repository, two methods are available. One could have a object, e.g. the method that is executed by a meta-filter, and request all attached annotations. The other option is to request the a DotNetType representing an annotations type and request all instances - the actual annotations - of that type.

Another file that is parsed during initialization is the filter specification, "filterdesc.xml". An example of this file is presented in Appendix A. It contains information about the the accept- and reject-actions of filter types, the operations performed by actions and a specification of constraints (conflicts and requirements). Figure 6.3 shows the data structure containing this information. The Repository singleton can be used to retrieve the FilterAction object belonging to the accept or reject of a certain filter type, as is specified in the FilterActionDescription objects. The FilterAction objects have a method *getOperations()* to get all operations performed by this action. The MetaAction is a special FilterAction which doesn't get it's operations from the filter specification file. During initialisation all *Semantic* annotations are fetched that are connected to the methods called by the meta filters in the repository. The string containing the operations is parsed and stored in a Hashmap as a list of operations with the Filter object as key. Whenever *getOperations()* is called with a Filter object as agument the corresponding operations are returned. The constraints specified in "filterdesc.xml" are represented by objects of type Constraint.

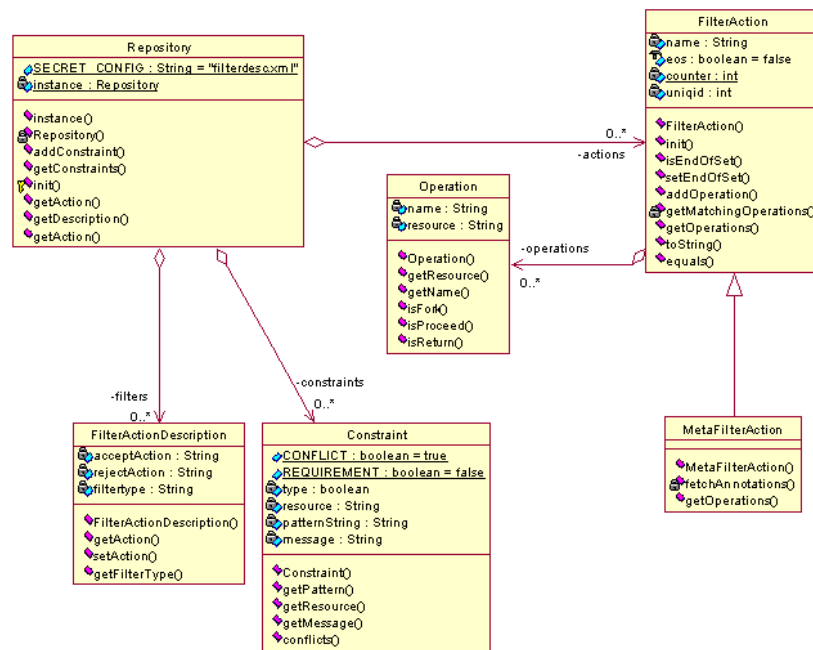


Figure 6.3: SECRET's private repository for filter semantics

6.4 Implementation of the reasoning engine

The classes responsible for the analysis performed by SECRET are displayed in figure 6.4. The entry-point of the tool is the *run()* method in class SECRET, which will iterate over all concerns in the repository and check if any filtermodules are superimposed on the Concern, because only these concerns require analysis. When one or more filtermodules are found a ConcernAnalysis object is created which has a method *checkOrder(FilterModuleOrder)* to check

a possible filtermodule-order for the concern. Depending on a property read from the project configuration-file SECRET can run in three different modes:

Normal: The selected filtermodule-order by FILTH (see chapter 2) will be checked.

Redundant: All possible filtermodule-orders are checked.

Progressive: Similar to *Redundant* mode all filtermodule-orders will be checked. However, if during analysis of the filtermodule-order selected by FILTH any conflicts are detected, the first filtermodule-order without conflicts will be selected instead. If there are more orders without conflicts, any of these could have been selected. The implementation of selecting the first one was however easiest to achieve.

In practice *redundant* mode has proven itself most useful, since reported conflicts are often just warnings telling the programmer there *might* be unexpected interference between two or more filters/filtermodules. If a warning represents an actual conflict is a matter of requirements.

The *checkOrder()* method of the ConcernAnalysis object creates a new FilterSetAnalysis object passing a list of filters constructed from the filtermodule-order. All possible executions are calculated. Then the executions that do not end with a dispatch-action or an error-action are removed. The number of paths through a list of n filters is 2^n because every filter can either accept or reject. Because we can filter out some executions the order of SECRET becomes $2^n - x$. Some impossible executions can not be filtered out. For instance, when two filters have identical matching-patterns they should always both perform their accept-action or reject-action. The matching-patterns of two filters can also match a disjunct set of messages. Then a filter should perform its reject-action when the other performs its accept-action. In an ideal world we would have FIRE analyze the filterset and use the resulting tree to fetch all possible executions of this filterset since this is what FIRE is designed for. However, FIRE does not support meta-filters.

Every execution, which consists of a list of filter actions, is then used to create an ExecutionAnalysis object and the *analyze()* method is then called which will return any detected conflicts. These ExecutionAnalysis objects are an implementation of the model described in chapter 5. By using secret's data-structure shown in figure 6.3, the list of actions is iterated resulting in a list of operations, which is then re-arranged as is shown in section 5.3. Then the operations are added to the corresponding resources and the constraint-specifications are matched against these resources. Since its implementation by [11] some additional specification-options have been added to these constraints. Constraints can be either conflicts and requirements. Conflicts cause a warning when the pattern matches. Requirements cause a warning when the pattern does not match. Constraints can also be applied to all resources by using a "*" for the name of the resource.

In chapter 5, the patterns have been specified with the complete names of the operations. The implementation and the actual pattern-specifications - also shown in the example XML specification in appendix A - only use the first character of an operations. One should be careful that no operations are used on a resource with an identical first character. Pattern-matching is case-sensitive.

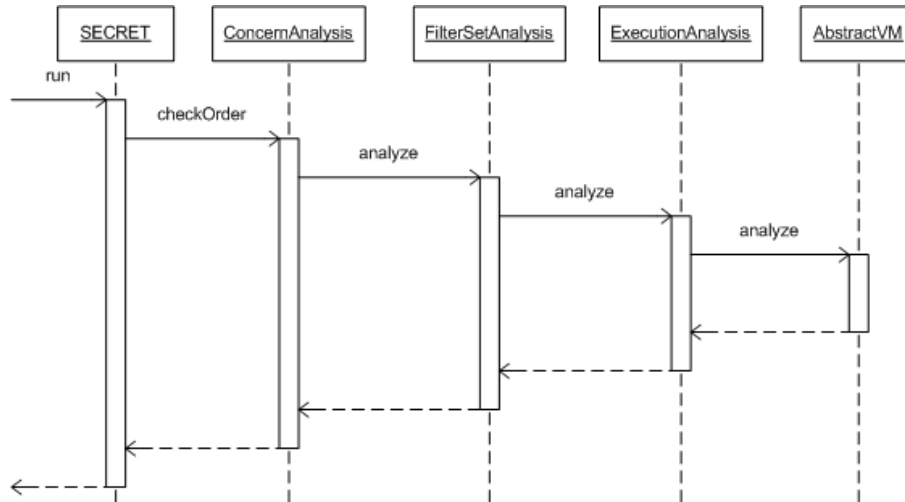


Figure 6.4: Sequence-diagram of SECRET's analysis phase

6.4.1 Reporting

Reporting conflicts is implemented in the *checkOrder(FilterModuleOrder)* method of class *ConcernAnalysis*. When a call to *analyze()* of a *FilterSetAnalysis* object returns any conflicts, these conflict are written to an HTML file. Figure 6.5 shows the generated report for the Jukebox example. The report-file is generated using SECRET's progressive mode. Every light gray box contains the analysis of a possible filtermodule-order. In such a box first the filterset is printed. Then, all executions containing conflicts are listed, with the actions performed in the execution on the left and the conflicts caused by the execution on the right.

As explained in chapter 3 there would be a conflict in the first reported filtermodule-order. In this case the song is played but no credits are withdrawn. This is the case for the executions that start with a meta-action performed by the enqueue-filter, the second and third execution. The rest of the actions do not matter. These filters are never reached since the enqueue-filter returns the message.

For the first execution an unexpected conflict is reported. The song is payed but not played. However, this execution is impossible. Both meta-filters have the same matching-pattern. Therefore they should either both perform a meta-action or both perform a continue-action. The second filtermodule-order, which we did not anticipate any conflicts in, has the same problem. The executions reported, and thus the conflicts, are impossible.

This means that we are still analyzing executions of a filterset. In the next section we explain how this can be avoided by coupling SECRET to the *Filter Reasoning Engine* (FIRE).

6.4.2 Coupling with FIRE

FIRE was designed as an engine offering an interface to other tools that need information about a set of filters. Internally, FIRE constructs a tree containing all possible executions of a filterset. Every path from the root-node to an end-node represents a the execution of a message with a

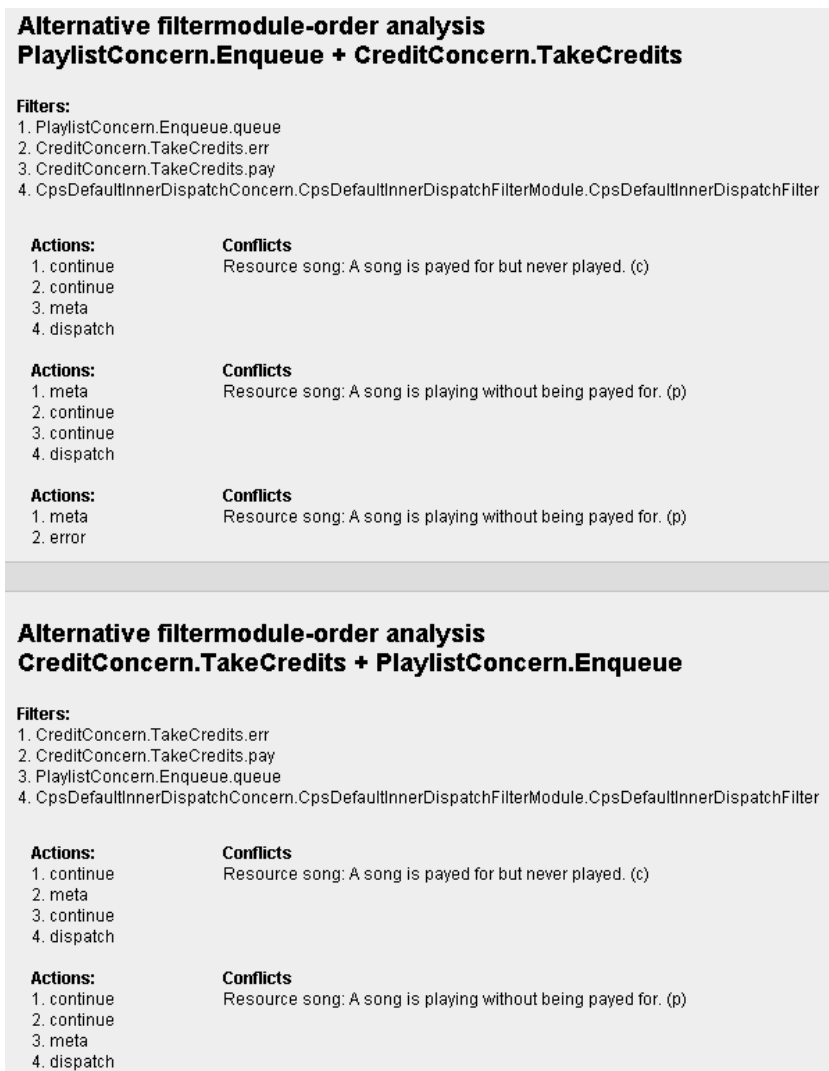


Figure 6.5: Report file of the analysis of the Song concern

certain set of conditions.

With this tree FIRE can tell exactly the executions that will happen in a certain state (conditions) for a all possible messages. FIRE keeps track of the target and the selector so we can also tell where a message will finally be dispatched. When SECRET would be coupled to FIRE we would only analyze executions that would actually happen and, in case of conflicts, we could even notify the developer for which messages and in what state the conflicts would occur. Sadly the meta-filter introduces some problems to FIRE.

Since FIRE does not have a clue what the effect will be of the meta-filter on the message, the tree will explode after every MetaAction into all possible combinations of the target, selector and conditions. Of course not every ACT changes the target and selector. Also, when an ACT replies the message, the MetaActionNode would become the end of a branch in the FIRE tree. Our first approach was to specify the required changes to the target, selector and conditions and use this to remove all impossible paths from the FIRE tree. However, the tree-explosion

was found to be more serious than expected. This is expressed by the order of the size of the FIRE tree, which is $O_F = (st2^c)^m$. In this formula, s is the number of selector symbols, t is the number of target symbols, c is the number of conditions and m is the number of meta-filters. Every meta-filter normally introduces one extra target-symbol and one extra selector-symbol, the internal or external act and the name of the method to call. We can do some prognosis on the increase of this order when the number of meta-filters rises. Without any conditions and just one meta-filter O_F is 6. For two meta-filters O_F has a value of 324 and when we increase the number of meta-filters to three O_F becomes 32768.

Another problem was found during analysis of FIRE. During the generation of the tree, FIRE performs a substitution of the target and selector in every filter before adding the filter action. The target and selector can then be read from the message and used in the filter action. In case of a substitute-filter the action would already be completed since the substitution has already been performed. A dispatch-filter would use the substituted target and selector to resolve the method to be dispatched to. The problem arises when we want to include a meta-filter. Identical to a dispatch-filter, the meta-action reads the target and selector from the message to be able to call the right method. However, after the ACT resumes the message, the target and selector should return to their original values, before the substitution is performed. However, due to the internal working of FIRE, this substitution is irreversible.

Until these issues with FIRE have been resolved SECRET will have to keep calculating the possible executions itself, taking a few impossible ones for granted.

6.5 Implementation of the Meta-filter run-time

When a program is executed that has been compiled with Compose*, every message that is subject to filters is intercepted and passed to a class called MessageHandlingFacility (MHF). This class will retrieve a unique ObjectManager object for the target object. The ObjectManager is the object that is responsible for handling all incoming and outgoing messages of an object. For every message the filters of the super-imposed filtermodules are evaluated. Depending on a filter accepting a message an accept- or reject-action is requested, from an object representing the filter's type, and executed. For instance, for an accepting dispatch-filter a DispatchAction will be executed. When a meta-filter accepts a MetaAction object will be executed.

6.5.1 Concurrency

Chapter 4 shows that the operations, that can be carried-out on the ReifiedMessage object, can cause concurrency between the sender object, the evaluation of the filterset, and the ACT. To be able to achieve this, a new thread is created for every ObjectManager handling the messages that should be filtered. We will refer to these threads as the *filterset-thread*. Also, every MetaAction will start a new thread which will run the ACT-method. We will refer to this thread as the *act-thread*. The thread where the message was sent from is referred to as the *sender-thread*.

To achieve the synchronization between these threads the class SyncBuffer is used that will lock, by calling *consume()*, whenever a value is requested. The call will be locked until a value is submitted to the buffer, by calling *produce()*. The *consume()* method is synchronized which

causes the buffer to operate as a FIFO.

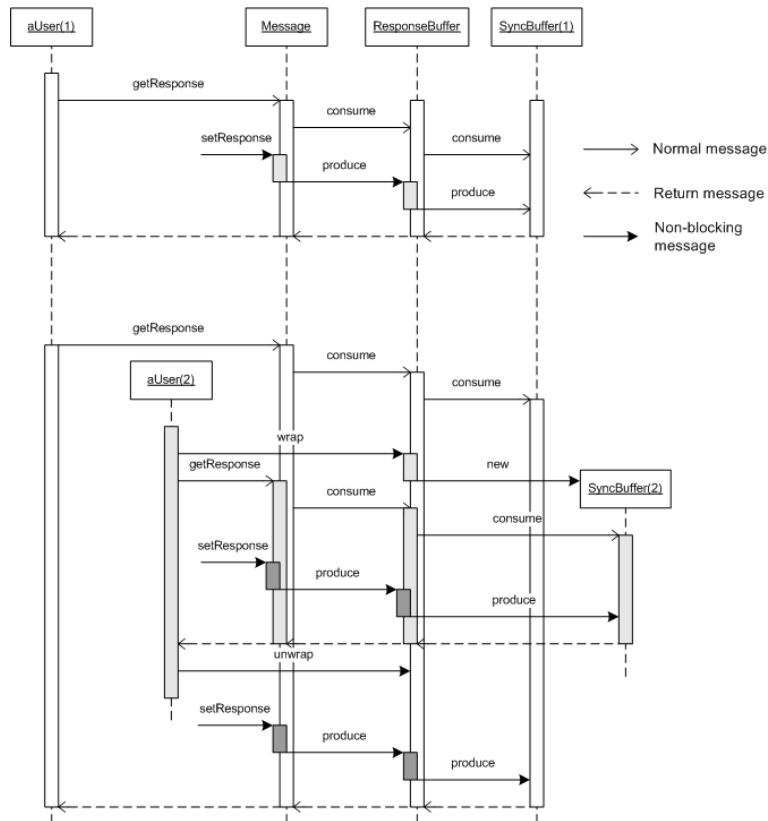


Figure 6.6: Response buffer usage

The `SyncBuffer` is used by the `ResponseBuffer` which is designed to supply the functionality we need when three different threads can set a response to and get the response from a message. The workings of the `ResponseBuffer` and the `SyncBuffer` classes are shown in figure 6.6. The objects `aUser(1)` and `aUser(2)` represent objects that will use the functionality explained here. The `ResponseBuffer` uses a stack of `SyncBuffer` objects. By manipulating this stack it is possible to manipulate the order of user-objects receiving a response.

In the first part of the sequence-diagram a response is requested from a message object by calling `getResponse()`. This method again calls `consume()` upon the message's `ResponseBuffer` which then calls `consume()` on the `SyncBuffer` on top of the stack. When `setResponse()` is called on the `Message` object, `produce()` will be called on the `ResponseBuffer` and finally on the same `SyncBuffer` on top of the stack, causing the `getResponse()` to return the produced value. The different colors indicate activity in different threads. When and by whom `setResponse()` is called is explained later in this section. What we want to show is that a call from another thread can unlock a thread waiting for a response from the `Message` object.

In the second part of the sequence-diagram, two methods are introduced to manipulate the stack. The `getResponse()` results in a `consume()` on the first `SyncBuffer`. Then, a second user-object calls the `wrap()` method. This method will create a new `SyncBuffer` and push it on top of the stack. A `getResponse()` call by the second user-object then results in a `consume()` on the newly created `SyncBuffer`. When `setResponse()` is called the value is again produced to the

SyncBuffer on top of the stack and *getResponse()* from the second user-object will return the produced value. By calling *unwrap()* the SyncBuffer is removed from the stack allowing another *setResponse()* call to return the *getResponse()* from the first user-object.

6.5.2 Message Queue

The evaluation of a message through a filterset is an atomic action. This means that when a message enters the filterset, other incoming messages are blocked until the first message is dispatched; as soon as the dispatched method starts executing, the next message is processed. This however implicates that a DispatchAction may not be executed in the filterset-thread because this thread should start processing the next message. Figure 6.7 illustrates the run-time process of processing a filterset with an accepting dispatch-filter. All white activity is executed in the sender-thread; the light-gray activity executes in the filterset-thread. Both threads run in the same ObjectManager instance so this object has two axes in the diagram to keep it readable.

In the sender-thread the MHF delivers the message to the ObjectManager, which produces the message to a message-queue - a SyncBuffer instance (remember the SyncBuffer behaves like a FIFO). Then the filterset-thread is notified and *getResponse()* is called which will block the sender-thread until *setResponse()* is called on the message object.

The *filterset-thread* is started by the *run()* call on another activation-axis of the same ObjectManager instance in the sequence-diagram. It retrieves a message from the message-queue by calling *consume()*. The accepting dispatch-filter will return a DispatchAction. However, if the DispatchAction would be executed now, the ObjectManager would still be locked and not be able to start processing the next message in the queue. Instead of calling *execute()* on the DispatchAction, the DispatchAction is passed as an argument to the *setResponse()* method of the message object. The caller-thread is unblocked and the *execute()* method is called by *getResponse()* method (if the response is of type DispatchAction). Meanwhile, the message processing thread can *consume()* the next message from the message-queue and start processing.

6.5.3 Including MetaActions in the run-time

As described above, the MetaAction will call a method in the ACT. However, we cannot wait until this message is completed to complete the execution of the MetaAction. Whenever *proceed*, *resume* or *return* is called on the ReifiedMessage object, the MetaAction should finish up and decide if the ObjectManager should continue to the next filter. Meanwhile, the execution of the ACT has not completed yet. The ACT has therefore to be executed in its own thread. We use a SyncBuffer, which we will refer to as the continue-buffer, to achieve synchronization between the ACT and the MetaAction. Figure 6.8 shows a sequence diagram of a MetaAction followed by a DispatchAction. The white and light-gray activity again correspond to the sender-thread and the filterset-thread, respectively. The dark-gray activity corresponds to code executed in the act-thread.

When *execute()* is called on a MetaAction the act-thread is starting with a *run()* call on the ReifiedMessage. This thread then executes the ACT-method. After starting the act-thread the

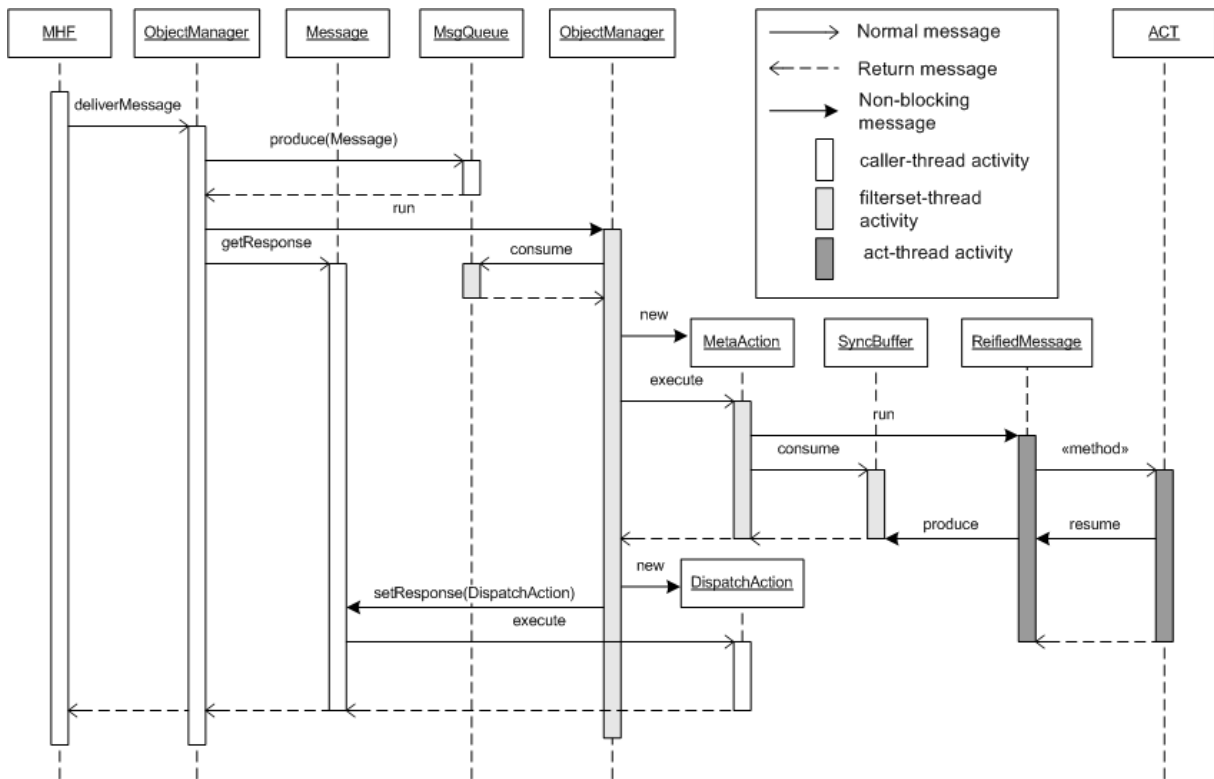


Figure 6.8: Sequence-diagram of a resuming ACT.

the return-value as argument.

Respond: Figure 6.11 is a sequence-diagram where first `respond()` is called followed by `proceed()` and an automatic `resume()` when the ACT finishes execution. The `respond()` call should re-activate the sender-thread. In the case illustrated in the diagram this could have been done by calling `setResponse()` on the Message object. However, the ACT might be preceded by another ACT that wrapped the ResponseBuffer by using `proceed()`. Therefore the `produceFirst()` method was added to the ResponseBuffer class, which will produce the SyncBuffer on the bottom of the stack instead of the buffer on top of the stack, causing the sender-thread to continue. Then `proceed()` is called and finally the DispatchAction is executed in the act-thread. If `resume()` would be used instead of `proceed()` the DispatchAction had to be executed by the sender-thread. However, after `respond()` the sender-thread was no longer waiting for a response so the DispatchAction would never execute. Therefore a `respond()` always has to be followed by a `proceed()`.

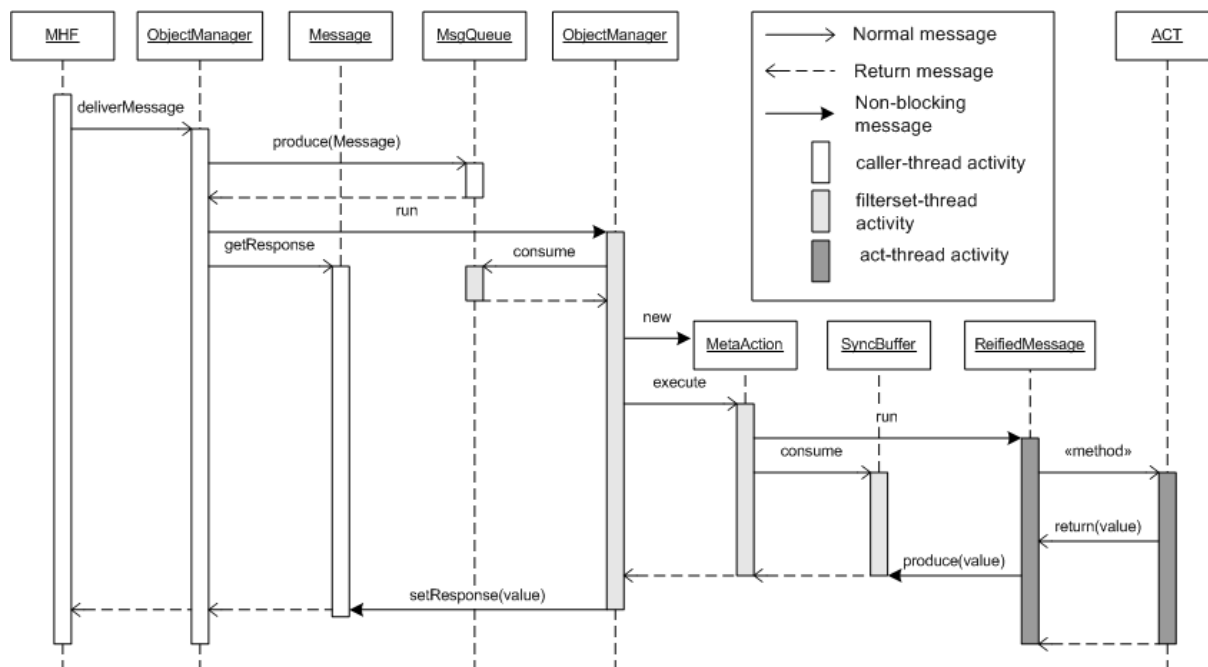


Figure 6.9: Sequence-diagram of a returning ACT.

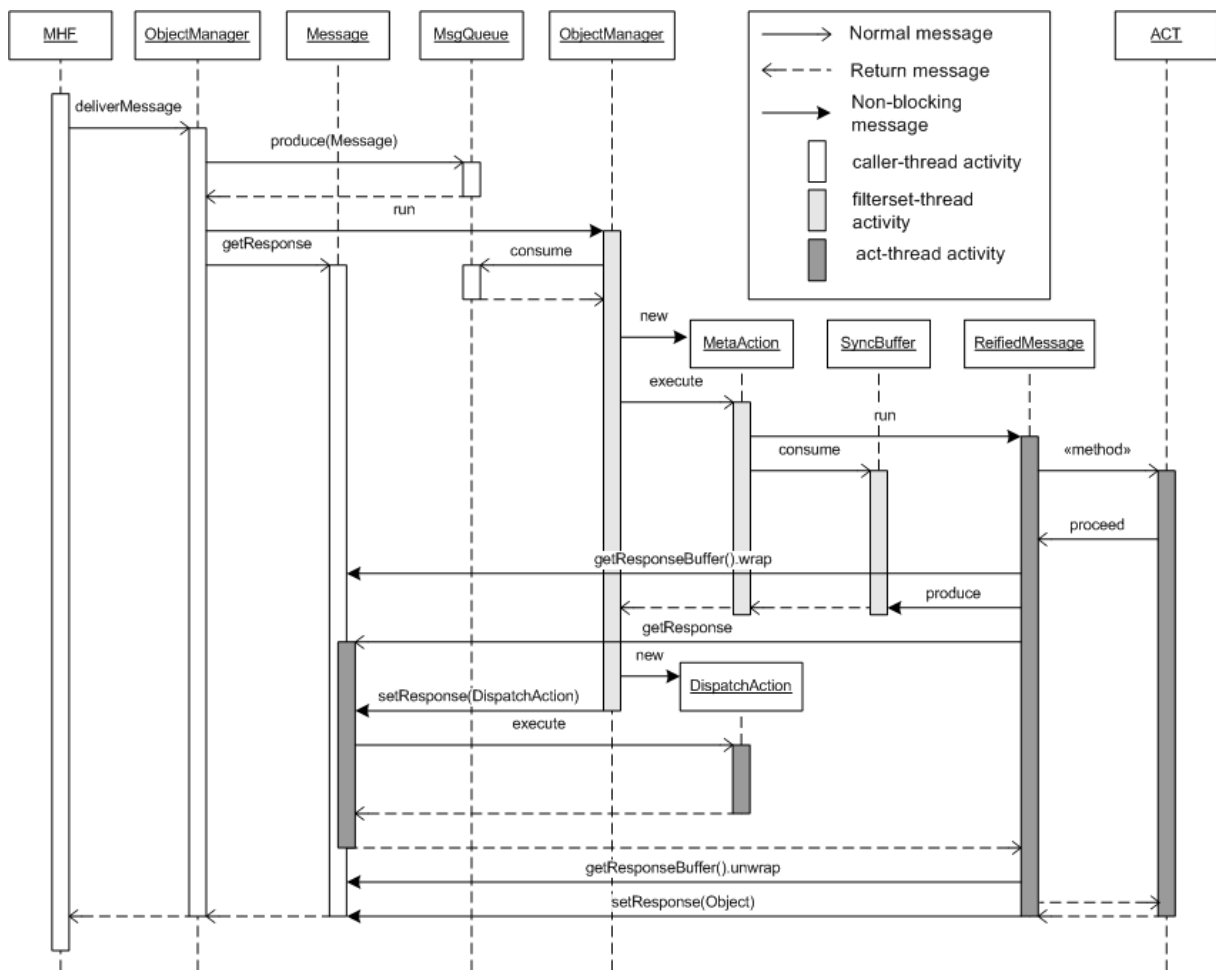


Figure 6.10: Sequence-diagram of a proceeding ACT.

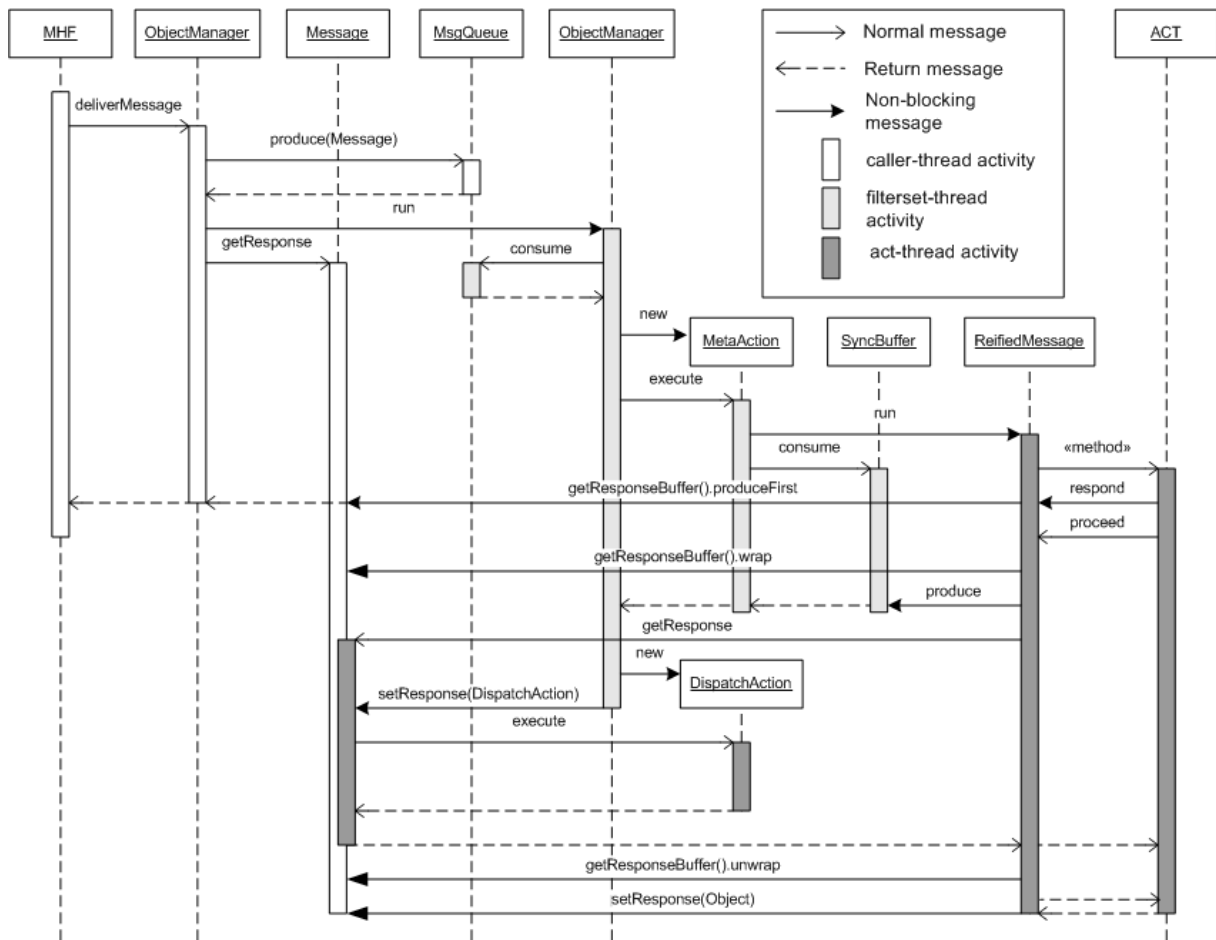


Figure 6.11: Sequence-diagram of a responding ACT.

Chapter 7

Conclusions

A conclusion is the place where you got tired of thinking.

7.1 Related work

The analysis of aspect oriented software is becoming more and more popular. The problem of aspect-interference has been widely acknowledged and several research projects address this issue. However, reasoning about the aspect code (advice) itself is very difficult in most AOP languages.

L. Blair and M. Monga [5] analyze possible aspect-interference in AspectJ programs. To analyze an aspect they consider the aspect itself and the part of the system it affects. The pointcut is used to compute a slice of the base-system. Non-interference at code level can then be guaranteed if the slices associated to different aspects are disjoint. This results in a mechanism for detecting shared join-points. Detection of actual conflicts at these join-points is not achieved.

The research of M. Störzer and J. Krinke [27] is also aiming at the detection of aspect-interference in AspectJ programs, but on a static level. Their approach identifies the impact of an aspect as the changes made to the base-system by changing the static structure of the base system, for instance by changing the inheritance hierarchy or introducing new members. A method is presented to decide *if* an aspect modifies base system behavior. Future work should allow detecting *where* the system behavior is influenced by an aspect.

7.2 Discussion & limitations

The scalability of SECRET has not been tested thoroughly. We can however deduce the order of execution-time of SECRET from its algorithm. Per concern, the maximum number of possible filtermodule-orders that have to be analyzed is $f!$, where f is the number of super-imposed filtermodules on the concern. When this reaches high levels in large projects one can

use SECRET's normal mode to limit analysis to one selected filtermodule-order per concern. The order of the analysis of a filtermodule-order depends on the number of possible executions, which is $2^n - x$, with n the number of filters and x the number of executions we can filter out. The order of analysis of one execution is linear to the number of filters. Thus, the total number of filters of the filtermodules super-imposed on a concern is critical for the order. A simple test has been done, with SECRET running in normal mode, by increasing the number of filters and counting the seconds SECRET took to analyze. The limit was found around 14 filters (not including the default dispatch to inner), which took around 30 seconds.

One should always remember that the report generated by SECRET contains *warnings* rather than *conflicts*. Due to the abstraction level of the resource-model, a specification of a conflict warns us merely that a conflict could be present in a certain execution. As was explained in chapter 6 the report currently also contains conflicts in executions that are impossible for a certain filterset, since we are unable to use FIRE in the elimination of impossible executions.

Only one single semantic specification of an ACT can be given. If for instance an *if-statement* is used around a statement with semantic relevance, there is no way to specify this. A developer should therefore try to keep his ACT as simple as possible, for instance by using conditions (declared in the filtermodule) instead of *if-statements*. If this does not work, the developer should remember to specify the worst-case scenario of the execution of the ACT. It is better to generate warnings that are not conflicts rather than not detecting a conflict at all.

7.3 Conclusion

In this thesis a solution has been presented to detect semantic conflicts between aspects that have turing-complete advice. The method has been implemented as part of Compose*, the .NET implementation of Composition Filters. However, the method can also be used for other aspect-oriented languages. In Compose*, it enables SECRET, the semantic reasoning tool, to reason about meta-filters. A meta-filter can reify a message and send it to a first-class object. These objects, which are called Advice Types (ACTs), are the way to create custom (turing-complete) advice in Composition Filters.

The behavior of a meta-filter is specified by annotating the invoked method in the ACT with a semantic specification of the method. This specification is a list of operations on abstract resources representing the semantics of the method.

The power of the conflict detection model depends largely on the detail of specification of an ACT. Therefore, a mapping has been presented of the methods of class ReifiedMessage to the resource-operations that should be specified by a developer of an ACT. New resources and operations have been introduced to capture the semantic capabilities of ACTs. Also, conflicts caused by the use of meta-filters have been identified and patterns to detect these conflicts have been presented.

The operations the ACT can execute on the reified message to control the message's execution have been evaluated and modified to fit possible needs of aspect-programmers.

SECRET has been modified to use the specifications from the annotated ACT when reasoning about meta-filters. Also, SECRET can optionally analyze all possible filtermodule-orders instead

of just the selected filtermodule-order of a concern. An option has been built in to allow SECRET to change the selected filtermodule-order when the selected order causes conflicts.

The filter-interpreter, which is used in the run-time environment of a Compose* project to intercept and evaluate message, has been modified to support meta-filters. A threading-model has been created that allows for all features provided to ACTs by a reified message.

7.4 Future work

Although we can already detect a fair amount of conflicts, some improvements, both small and large, can be made to the reasoning tool. The following subsections present suggestions for future work.

7.4.1 Alphabet checking

Currently there is no checking mechanism for the operations used on resources. This means that a typo in a specification does not raise a warning but results in a new resource or operation being added. Also, for the conflict-patterns, the first character of the name of the operations is used. Therefore it is not possible to use operations with identical first characters. The declaration of all resources and allowed operations is suggested, together with a mechanism to check for naming collision and typos.

7.4.2 Respond a proxy

When *respond()* is used in an ACT, the sender of the method continues before the called method is invoked. We can however allow the sender of the message to request the return-value of the method by responding with a *Proxy* object, as presented in [23]. As soon as the sender requires the return-value it can be requested from the Proxy, which will block the sender until the return-value is available. The implementation of the Proxy class uses a wait-filter to wait for the return-value to become available. Currently, the wait-filter is not supported by the filter-interpreter.

7.4.3 Filter impossible executions

As mentioned, SECRET calculates the possible executions of a filterset itself, using a simple strategy to remove impossible executions and duplicate executions. However, there is still a possibility of executions being analyzed for semantic conflicts where these executions will never take place. FIRE however reasons about the filter specification and keeps track of message state and message modifications. Therefore, FIRE knows exactly which executions will be possible and what method will finally be dispatched to. However, to use the full power of FIRE it will be necessary to supply some information about the ACT to FIRE. For example, if the target or selector is modified before the message is resumed, FIRE should know about this. This also calls for finding a solution to the problems discussed in section 6.4.2.

7.4.4 Annotate dispatched methods

When an ACT uses *proceed()* to continue a message, part of the ACT executes after that message is dispatched. It might therefore be useful to also be able to specify the semantics of the invoked method in the reasoning process. Currently we can only specify generic behavior of a method invoked by a dispatch-filter. This it is specified at the dispatch-action. Of course it does not sound tempting to specify the semantics of every method in a program. SECRET could warn a developer which methods should be annotated with a semantic specification because they are invoked after a *proceed()*. When FIRE fully supports meta-filters, we can resolve which method will be invoked when a message is dispatched.

7.4.5 Comparing filtermodule-orders

The current implementation of SECRET only supports analysis of a single filtermodule-order. One could imagine that two different filtermodule-orders do not cause any conflicts, but result in programs with different behavior. For example, imagine a message sending a string, which is filtered by two meta-filters. One meta-filter modified the string. The other meta-filter logs the message, including the string. Depending on the order of the filters, the original string or the modified string is logged. The programmer probably intended one of the possible orders but neither filtermodule-orders have any conflicts. A solution to detect these differences could be to compare the sequences of operations for different filtermodule-orders. Unfortunately not all differences are relevant. Therefore, certain operations - or sequences of operations - should be identified as relevant. We could warn a programmer if a sequence of operations is performed in an execution of one filtermodule-order but not in another. Again we need FIRE, to be able to compare executions that correspond to the same message and state (conditions).

Bibliography

- [1] ada. Ada for the web, 1996. URL http://www.acm.org/sigada/wg/web_ada/.
- [2] M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. "Discussing Aspects of AOP". *Communications of the ACM*, Volume 44(issue 10), October 2001.
- [3] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. svlnes, 1993. URL <http://trese.cs.utwente.nl/publications/paperinfo/AbstrObjIntUsingCF.pi.top.htm>.
- [4] L. Bergmans and M. Aksit. Composing crosscutting concerns using Composition Filters. *Communications of the ACM*, Volume 44(issue 10), October 2001.
- [5] L. Blair and M. Monga. Reasoning on aspectj programmes. In *AOSD-GI Workshop*, pages 45–49, 2003.
- [6] P. S. Caro. Adding systemic crosscutting and super-imposition to composition filters. 2001.
- [7] M. Corporation. "Introduction to .NET, Hello World, and a Quick Look Inside the .NET Runtime". Technical report, Microsoft Corporation, apr 2003. URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnguinet/html/drguinet0_update.asp.
- [8] M. Corporation. ".NET Compact Framework - Technology Overview". Technical report, Microsoft Corporation, 2003. URL <http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/overview/default.aspx>.
- [9] M. 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>.
- [10] M. 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/cpconwhatiscomonlanguagespecification.asp>.
- [11] P. Durr. Detecting semantic conflicts between aspects. Master's thesis, University of Twente, 2004. URL http://janus.cs.utwente.nl:8000/twiki/pub/Composer/SECRET/Master_Thesis_Pascal_Durr.pdf.
- [12] T. Elard, R. E. Filman, and A. Bader. "Aspect-Oriented Programming". *Communications of the ACM*, Volume 44(issue 10), October 2001.

- [13] G. Kiczales and E.Hilsdale and J. Hugunin and M. Kersten and J. Palm and W. Griswold. "An overview of AspectJ". In *Proc. of ECOOP 2001*, 2001.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Programming language concepts and paradigms*. Addison Wesley, 1995.
- [15] M. Glandrup. Extending c++ using the concept of composition filters. 1995.
- [16] J. D. Gradecki and N. Lesiecki. "Mastering AspectJ: Aspect-Oriented Programming in Java". Wiley, 2003.
- [17] 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>.
- [18] jython. Jython homepage. URL <http://www.jython.org/>.
- [19] S. Katz. Diagnosis of harmful aspects using regression verification. In *Foundations of Aspect Languages (FOAL) Workshop*, 2004.
- [20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. proceedings of the european conference on object oriented programming (ecoop). 1997.
- [21] H. Kim. *AspectC#: An OASD implementation for C#*. PhD thesis, Trinity College Dublin, 2002.
- [22] P. S. Koopmans. On the definition and implementation of the Sina/ST language. Master's thesis, University of Twente, 1995.
- [23] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.
- [24] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and the Hyperspace Approach". In *Software Architectures and Component Technology: The State of the Art in Research and Practise*. Kluwer Academic Publishers, 2001.
- [25] A. Popovice, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. 2002.
- [26] A. Popovice, T. Gross, and G. Alonso. Just-in-time aspects: Efficient dynamic weaving for java. 2003.
- [27] M. Storzer and J. Krinke. Interference analysis for aspectj. In *Foundations of Aspect Languages (FOAL) Workshop*, 2003.
- [28] D. Stutz. The Microsoft Shared Source CLI Implementation. 2002.
- [29] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*, 2000.
- [30] 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>.
- [31] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.
- [32] J. C. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the java language. 1999.

Appendix A

SECRET Configuration file

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <secret>
3   <filters>
4     <filter type="Error">
5       <accept action="continue"/>
6       <reject action="error"/>
7     </filter>
8     <filter type="Dispatch">
9       <accept action="dispatch"/>
10      <reject action="continue"/>
11    </filter>
12    <filter type="Meta">
13      <accept action="meta"/>
14      <reject action="continue"/>
15    </filter>
16    <filter type="Substitute">
17      <accept action="substitute"/>
18      <reject action="continue"/>
19    </filter>
20    <filter type="Wait">
21      <accept action="continue"/>
22      <reject action="wait"/>
23    </filter>
24  </filters>
25
26  <actions>
27    <action name="continue"/>
28    <action name="meta"/>
29    <action name="error" endofset="true">
30      <operation name="error" resource="message"/>
31      <operation name="lock" resource="returnvalue"/>
32      <operation name="return" resource="message"/>
33    </action>
34    <action name="dispatch" endofset="true">
35      <operation name="write" resource="target"/>
36      <operation name="write" resource="selector"/>
37      <operation name="read" resource="args"/>
38      <operation name="write" resource="returnvalue"/>
39      <operation name="dispose" resource="args"/>
40      <operation name="dispose" resource="target"/>
41      <operation name="dispose" resource="selector"/>
42      <operation name="return" resource="message"/>
43    </action>
44    <action name="substitute">
45      <operation name="write" resource="target"/>
46      <operation name="write" resource="selector"/>
47    </action>
```

APPENDIX A. SECRET CONFIGURATION FILE

```

48     <action name="wait">
49         <operation name="write" resource="timing"/>
50     </action>
51 </actions>
52
53 <constraints>
54     <conflict
55         resource="message"
56         pattern="R(.*?)R"
57         message="The message is responded twice." />
58     <conflict
59         resource="message"
60         pattern="p(.*?)r"
61         message="Proceeded message returns an exception." />
62     <conflict
63         resource="returnvalue"
64         pattern="^r"
65         message="A return-value should be written before it is read." />
66     <conflict
67         resource="message"
68         pattern="R(.*?)R"
69         message="The message is responded twice." />
70     <conflict
71         resource="*"
72         pattern="l.*"
73         message="Locked resources can not be read or written." />
74     <conflict
75         resource="*"
76         pattern="w$"
77         message="The resource is written but changes are never used." />
78     <conflict
79         resource="*"
80         pattern="d.*w"
81         message="A song is played but not credited." />
82     <!-- We use the two seperate conflicts instead of the requirement below
83     <require
84         resource="song"
85         pattern="^(cp|pc)?$"
86         message="A song is either credited or played, where both should happen." />
87     <conflict
88         resource="song"
89         pattern="^p$"
90         message="A song is playing without being payed for." />
91     <conflict
92         resource="song"
93         pattern="^c$"
94         message="A song is credited but not played." />
95 </constraints>
96
97 </secret>

```