Detecting semantic conflicts between aspects

(in Compose*)

Ing. P.E.A. Dürr

April 2004

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

Prof. dr. ir. M. Aksit
Dr. ir. L.M.J. Bergmans
Dr. ir. A. Rensink
I. Nagy MSc.
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.

The Microsoft .NET framework is centered around the Common Language Infrastructure. All .NET supported languages are mapped to one common intermediate language. The result is a language independent programming environment where one language can call functions and methods of other languages.

The main goal of the Compose* project is to incorporate the composition filters on the .NET intermediate language. With one effort all .NET supported languages are thus made aware of aspects. The combined result is a language independent aspect oriented approach on a language independent programming environment. For instance an aspect written in C# and being imposed on a FORTRAN base system.

To impose multiple aspects in one system may introduce new problems. These aspects can be imposed on the same joinpoint. Such a joinpoint is then shared between multiple aspects. There could be ordering constraints, stating which aspects should be executed before the others. However even this cannot shield us from the problem that multiple aspects may conflict. Such a conflict may reduce the functionality of the aspects, but even worse, it may reduce the functionality of the underlying system. Conflicts can be divided into two areas. One where a message cannot pass because the aspects excludes its execution. The other is more semantic or pragmatic. They conflict in the purpose or side-effects of the aspects.

This thesis presents a solution for detecting the latter kind of conflicts. In order to detect such conflicts a higher level of abstraction is needed. An abstract model is used to encapsulate the areas where aspects conflict. The transformation from aspects to a model is extremely difficult when aspects are expressed in a full programming language like AspectJ. In this case one has to be able to reason about the actual source code and to extract the relevant information.

In the composition filters approach, aspects are declaratively defined. The transformation from this source code to the model is therefore far more easier than in other aspect oriented ap-
approaches. This allows the creation of a tool which is able to automatically reason about the semantics of filters in given some specification.

The presented model consists of two main parts. Every filter executes certain actions on an Abstract Virtual Machine. In the case of composition filters these are for instance “continue to next filter” or “wait until condition is valid”.

Inside the virtual machine the actions are translated into operations on resources. These resources are the concrete or abstract areas where the filters conflict. After all filters in a given filter set are evaluated, all resources have a sequence of operations. This sequence is subsequently matched against a regular expression to identify if there is a conflict or not.

The entire model is expandable and user adaptable. The user only needs to specify which actions are translated to operations on certain resources and what the non conflicting regular expressions are for those resources.
Acknowledgements

First of all I would like to thank dr. ir. Lodewijk Bergmans. In the summer of 2003 he took me under his wings and opened my eyes to the world of research and science at ECOOP2003. He also “lighted” my interest in aspect orientation with his passion. Without him I might not have applied for the PhD position at the university. Furthermore I would like to thank dr. ir. Arend Rensink for his guidance and patience. With his mathematical background and exhaustive knowledge he forced me to think beyond the obvious. I would also like to thank prof. dr. ir. Mehmet Akşit and Istvan Nagy MSc., although they were mostly in the background, they did influence my work.

Secondly I would like to thank the Compose* team members. The meetings were always a pleasure with you guys around. I would like to mention Frederik Holljen en Raymond Bosman. We shared many moments of laughter and despair throughout my time here at the University.

I would like to thank my roommates at the Campuslaan 39. The ever pleasant moments late at night while I was working were a source of both inspiration and agony. Though I may have neglected my duties as a roommate, I promise to make this up in the future.

A special thanks goes out to the people that reviewed my chapters. Some of them have already been mentioned, the others however not. These people were: Daphne de Klerk, Niels Zijlstra, Gürcan Gulesir and Ruben Wassink.

Finally, my gratitude goes out to my family, specially my father. His ever lasting patience and guidance have definitely increased the quality of the work and of the report. I will remember the discussions we had and I hope you will continue to offer a place for late night discussions. And the rest of the family which forced me to stay focused during dark times.
Abstract

Acknowledgements

1 Introduction

1.1 Introduction to Aspect-Oriented Software Development

1.1.1 The object-oriented approach to software development

1.1.2 Problems with the object-oriented approach

1.1.3 Example

1.1.4 The AOP solution

1.1.5 AOP composition

1.1.6 Aspect weaving

Source code weaving

Intermediate Language weaving

Adapting the Virtual Machine

1.1.7 AOP approaches

The AspectJ approach

The Hyperspaces approach

1.2 Composition Filters

1.2.1 The Composition Filters approach
2 Compose*

2.1 Introduction ................................................. 17
2.2 Overview of the .NET architecture ......................... 18
2.2.1 Features of the Common Language Runtime .............. 19
2.2.2 The .NET Framework class library ...................... 20
2.2.3 Standardization .......................................... 20
2.2.4 A comparison between the .NET CLR and the Java VM .... 23
2.3 Features explicit to Compose* ............................. 23
2.4 Demonstrating example ..................................... 24
2.4.1 The object-oriented design .............................. 25
2.4.2 Completing the Pacman example ...................... 26
2.5 Architecture ................................................. 28

Master ......................................................... 28
TYM (TYpe Manager pass one) .................................. 30
SUPRE (SUPERimposition PREprocessing) ...................... 30
REXREF (Resolve EXternal REферences) ......................... 30
SANE (Superimposition ANalysis Engine) ....................... 30
FILTH (FILTer composition & cHecking) ....................... 30
FIRE (FIlter Reasoning Engine) ............................... 31
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
7.3 The road ahead ............................................................... 81
  7.3.1 Detailed look at current filters ................................. 81
  7.3.2 Focus in the conflict reports ................................. 81
  7.3.3 Increase reasoning power .................................. 81
  7.3.4 Traversing down a dispatch chain ......................... 82
  7.3.5 Development of new filters ................................. 83
  7.3.6 Analysis of dynamic behaviour .............................. 83
  7.3.7 Introduce concurrency ........................................ 83
  7.4 Conclusion ........................................................... 84

Appendices ........................................................................ 87

A Example XML filter specification ................................. 89
Nomenclature

AOP  Aspect-Oriented Programming
API  Application Programming Interface
CIL  Common Intermediate Language
CLI  Common Language Infrastructure
CLR  Common Language Runtime
CLS  Common Language Specification
CONE  Code Generation
CORBA  Common Object Request Broker Architecture
CORE  Code Generation
CTS  Common Type System
FILTH  Filter Composition & Checking
FIRE  Filter Reasoning Engine
FLIRT  Filter Interpreter
GUI  Graphical User Interface
IL  Intermediate Language
ILICIT  Interception Inserter
JIT  Just-in-time
JVM  Java Virtual Machine
MSIL  Microsoft Intermediate Language
OOP  Object-Oriented Programming
PDA  Personal Digital Assistant
POP  Post-Object Programming
REXREF  Resolve External References
SANE  Superimposition Analysis Engine
SECRET  Semantic Reasoning Tool
SIGN  Signature Generation
SUPRE  Superimposition Preprocessing
TYM  Type Manager
UML  Unified Modeling Language
XML  eXtensible Markup Language
Introduction

“The comatose bride”

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

Consistency analysis and reasoning with composition filters.
Raymond Bosman

Christian A. Vinke
This thesis first introduces Aspect Orientation and illustrates the working of Composition Filters. The second chapter deals with the Microsoft .NET environment and its use in the Compose* toolset. The third chapter identifies the problems where a semantic reasoning is needed. The fourth chapter presents a model where the filters are analysed. The chapter also presents a formal model for the detection of semantic conflicts between aspects. Chapter five analyses the current filter types and provides the input of the model for the current filter types. Subsequently chapter six will briefly discuss the implementation effort. Finally the conclusions are stated and possible future extensions are discussed in chapter seven.

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 [37]. 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.
1.1 Introduction to Aspect-Oriented Software Development

Prolog is currently the most used logic language [37].

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 [37].

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 [22, 23]. 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 Ossher [28] 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 they 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.

```
public class A {
    private LogWriter log;
    public int a, b;

    A() {
        log = new LogWriter();
    }

    public void addTwoIntegers() {
        private int result;
        result = a + b;
    }
}
```
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 [10]. 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 they 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.

```java
aspect Logging {
  LogWriter log = new LogWriter();
  pointcut log (): call(A.addTwoIntegers());
  after (): log () {
    log.write(calculation performed);
  }
}
```

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 weaved 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 also has 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 added avoiding re-compilation, re-deployment, and re-start of the application [29, 30].

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 provide 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 [28], which together with Composition Filters [3] are today’s main AOP techniques. A detailed description of Composition Filters will be given in section 1.2.
1.1 Introduction to Aspect-Oriented Software Development

The AspectJ approach

AspectJ [15] 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:

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

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

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

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

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”.

Pascal Dürr
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:

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

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 [28] 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 [35] 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:
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:

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

```
hypermodule Pacman_Without_Debugging
  hyperslices: Feature.Kernel, Feature.Logging
  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 "mergeByNames" 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 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.

```java
class SmartGhost {
    filtermodule ghostMovement {
        internals
            ghost : Ghost;
        externals
            pacman: Pacman;
        methods
            int getFleeMove();
            int getHuntMove();
        conditions:
            pacman.isEvil;
        inputfilters
            move: Substitute = {
                isEvil => [ghost.getNextMove] inner.getFleeMove,
                True => [ghost.getNextMove] inner.getHuntMove );
            disp: Dispatch = { inner.*, ghost.* }
        }; 
    }
    implementation begin in "Java";
    public class SmartGhostImpl {
        public int getFleeMove() {
            return fleeMove;
        }
        public int getHuntMove() {
            return huntMove;
        }
    }
}
```

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.

```plaintext
concern Tracing {
    filtermodule tracingModule {
        externals
            log: Log;
        inputfilters
            logIn: Meta = ( isEnabled => [*,*] log.traceMessage );
        outputfilters
            logOut: Meta = ( isEnabled => [*,*] log.traceMessage );
    };
    superimposition {
        selectors
            withTracing = { ==Pacman, ==Ghost, ==World };
        filtermodules
            withTracing <= tracingModule;
    };
    implementation begin in "Java";
    public class Log {
        public void traceMessage(Message m) {
            // tracing functionality here
            // ...
            // continue evaluating this message
            m.fire();
        }
    }
};
```

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 [24].

1987: Together with Anand Tripathi of the University of Minnesota the Sina language was
further developed. The semantic network approach was replaced with declarative specifi-
cations and the interface predicate construct was added.

1991: The interface predicates where replaced by the dispatch filter and the wait filter took
over the synchronization functions of the object manager. Message reflection and real-time
specifications where handled by the meta filter and the real-time filter [25].

1995: The Sina language with Composition filters was implemented using Smalltalk [24]. 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 [38] was developed and implemented.
The implementation consisted of a preprocessor capable of translating Composition Filter
specifications into the Java language.

2001: ConcernJ [5] 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.
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 [6]. 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 [8]:

- To provide a consistent object-oriented programming environment where object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- 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 [8]: 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) [36]. 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 imple-
2.2 Overview of the .NET architecture

The implementation of the Common Language Infrastructure (CLI) and in this context the CLR is simply called the .NET Runtime or Runtime for short.

![Figure 2.1: The context of the .NET Framework. Source: Overview of the .NET Framework [8].](image)

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 CLRs for any platform, creating a true interoperability infrastructure [36]. 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\footnote{1}. At the time of writing, the .NET framework is the only advanced Common Language Infrastructure (CLI) implementation available. A shared-source\footnote{1} implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor\footnote{33}. 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.

\subsection{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.

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:

\begin{itemize}
  \item Console applications;
  \item Windows GUI applications (Windows Forms);
  \item ASP.NET applications;
  \item XML Web services;
  \item Windows services.
\end{itemize}

\subsection{Standardization}

The entire CLI has been documented, standardized and approved\footnote{18} by the European association for standardizing information and communication systems, Ecma International\footnote{2}. Benefits of this standardization for developers and end-users are:

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

\footnote{1}Only non-commercial purposes are allowed.

\footnote{2}An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at www.ecma-international.org.
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 [9]. Figure 2.2 shows the relationships between the CTS, the CLS, and the types available in C++ and C#.

In this way the standardized CLI provides, in theory\(^3\), 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 languages. This wide coverage demonstrates the power of the standardized CLI.

Figure 2.3 shows the relationships between all the main components of the CLI. The top of the figure shows the different programming languages with compiler support for the CLI. Because 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 library is also available to the developers. Finally, the integration of the CLR with the platform it is running on is shown.

---

\(^3\)Unfortunately Microsoft didn’t submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.
Figure 2.3: 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.
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) [21] 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 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 scenarios 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.

![Figure 2.4: The UML class diagram of the object oriented Pacman game.](image)

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 over 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.

### 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.

```java
concern SmartGhost {
    filtermodule ghostMovement {
        internals
            ghost: Ghost;
        externals
            pacman: Pacman;
        methods
            int getFleeMove();
            int getHuntMove();
        conditions:
            pacman: isEvil;
        inputfilters
            move: Substitute = {
                isEvil => [ghost.getNextMove] inner.getFleeMove,
                True => [ghost.getNextMove] inner.getHuntMove);
            disp: Dispatch = { inner.*, ghost.* }
        };

    implementation begin in "Java";
    public class SmartGhostImpl {
        public int getFleeMove() {
            // return best flee move
            return fleeMove;
        }

        public int getHuntMove() {
            // return best hunt move
            return huntMove;
        }
    };
}
```

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*.

Pascal Dürr
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. It intercepts each eat call and sends the message in a reified form to the eat methods in the implementation.

```java
... filtermodule scoreModule
{
  internals
  impl: ScoreImpl;
  externals
  game: Game;
  methods
  eatFood(int, int);
  eatVitamin(int, int);
  eatGhost(Ghost);
  roundOver();
  gameInit();
  inputfilters
  food: Meta = ([*.eatFood] impl.eatFood);
  vita: Meta = ([*.eatVitamin] impl.eatVitamin);
  kill: Meta = ([*.eatGhost] impl.eatGhost);
  round: Meta = ([*.roundOver] impl.roundOver);
  reset: Meta = ([*.gameInit] impl.gameInit);
}
... 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.

```java
... filtermodule scoreView
{
  internals
  impl: ScoreImpl;
  externals
  view: View;
  methods
  paint(Graphics);
  paint(Message);
  outputfilters
  paint: Meta = ([*.paint] impl.paint);
}
... 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.

```
... superimposition
{
    selectors
    view = { *=View };

    filtermodules
    scoring <- scoreModule;
    view <- scoreView;

};
...```

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.
Figure 2.6: The Compose* architecture
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 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 superimposition 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.
output: The 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.

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.
As discussed in the first chapter the use of aspect orientation can greatly increase the separation of concerns and the evolvability of current systems. AOP helps to identify crosscutting behaviour and presents tools and constructs to encapsulate these crosscutting concerns into reusable components. The concerns are applied to the code-base to create the desired functional system. According to the composition filters approach these concerns can be standalone objects or they are superimposed on already present system elements. This chapter will show some examples of problems of when composing aspects. The next section introduces a scenario based on the Snow White [17] fairytale. Firstly a basic object-oriented framework is introduced. Secondly some aspects are added to this framework and the problems introduced by the composition of these aspects are discussed.
3.1 Snow White example

“...so beautiful, even in death, that the dwarfs could not find it in their hearts to bury her... they fashioned a coffin of glass and gold, and kept eternal vigil at her side... The Prince, who had searched far and wide, heard of the maiden who slept in the glass coffin. She remains there through a full year, a cycle of seasons, as they stand around it grieving. Finally, her Prince comes and is relieved to find the ragged maiden that he had fancied at the castle. He gently kisses her cold red lips for farewell, not knowing that his Love’s First Kiss will awaken her from her deathlike slumber. With great joy and cheering in the forest, and they lived happily ever after...” [17].

As the main purpose of this thesis is not to describe the entire Snow White fairytale, only a small subset is chosen. The example uses the people in the fairytale to create a setting where certain problems with these persons arise. The problems are subsequently solved using aspects. These problem solving aspects introduce new conflicts. These conflicts are subsequently analyzed. The class hierarchy that is used in this example is shown in figure 3.1.

![Diagram of the Snow White tale](image)

Figure 3.1: Diagram of the Snow White tale

The hierarchy first introduces a class called `FairytaleCharacter`, this defines the general behaviour inherited by all characters in the fairytale. `SnowWhite`, `Prince` and `Dwarf` extend from `FairytaleCharacter`, so that they have the the general functionality of `FairytaleCharacter` plus extra functionality for their specific needs and actions. For instance the dwarfs have to be able to `work`, likewise `SnowWhite` has to be able to `cook`.

The conflicts that this thesis detects are found in any AOP approach. These conflicts are not solely found in the Composition Filters AOP approach. The aspects presented in this chapter are therefore given in `Compose*` and `AspectJ`.

3.2 Adding aspects

3.2.1 Keeping down the weight

The basic framework has now been explained. As obesity is becoming more and more of a problem, we want to prevent the people in the tale from getting over weighted. To prevent obesity we add an aspect to the `FairytaleCharacter` class, and its subclasses, to intercept every
call to the eat method. The aspect, Diet, checks the amount of food eaten by a person per day. If the total amount of food reaches six, all subsequent calls to the eat method are rejected. The source code of this aspect is given in both Compose* syntax and AspectJ syntax. These are presented in listing 3.1 and 3.2.

Listing 3.1: Diet aspect in Compose*

Listing 3.2: Diet aspect in AspectJ

3.2.2 Keeping Snow White healthy

The prince is concerned about Snow White as she continues to reduce the amount of food eaten. He wants to make sure his future bride is still healthy when they marry. He therefore decides to add an aspect to Snow White to make sure she eats sufficient. This aspect states that Snow White has to eat more than 10 units of food. The source code of this aspect is given in listings 3.3 and 3.4.
Figure 3.2 shows the original object-oriented framework with the aspects Diet and EnoughProtection added.

At a first glance there may not be a problem here, however the Diet aspect is also applied to SnowWhite via inheritance, which results in two aspects being applied to the same point. This raises the question which aspect is applied first. Figure 3.3 illustrates the two possibilities.

There are two possible orderings. In the first ordering the EnoughProtection aspect is executed and then the Diet aspect. In the second ordering the Diet aspect is executed before the EnoughProtection aspect. Both orderings yield a problem.

The condition for the EnoughProtection yields true if the total amount of food to eat is more than 10. The Diet aspect uses a condition which is true if the total amount of food is less then or equal to 6. The conditions are disjoint, if one is true the other is definitely false and
vis a versa. These kinds of conflicts can not be detected by looking at one filter specification. These conflicts can only be detected by reasoning about the semantics of the conditions. These conditions can be implemented in any .NET supported language. This allows the conditions to be very expressive. It does however reduce the ability to reason about them.

This example illustrates the situation where two combined aspects create a conflict. Both aspects function correctly on their own. The combination, in a certain order, has unintended side-effects. The example also illustrates that if aspects are added at a high level in the inheritance tree, programmers working on aspects for the lower level objects may not be aware of these high level aspects. Even in this trivial small example such problems can be overlooked. In even larger applications these problems occur more easily and are often unnoticed.

3.2.3 Dinner in time

This sections introduces two new aspects. The \textit{EatBeforeSix} aspect states: “diner should be ready at six o’clock”. The dwarfs deserve a well prepared and warm meal when they return from working in the mines. Again the source code of this aspect is shown in both Compose* and AspectJ. These are shown in listings \ref{lst:eatbeforesix-compose} and \ref{lst:eatbeforesix-aspectj}.

```plaintext
concern EatBeforeSix begin
  filtermodule seteatetime begin
    inputfilters
    settime : RealTime = { True => [ *.cook ] message.deadline=1800 };
  end filtermodule
  superimposition begin
    selectors
    snowwhite = { SnowWhite };
    filtermodules
    snowwhite <- seteatetime;
  end superimposition
end concern
```

Figure 3.2: Diagram of the Snow White tale, with aspects

Figure 3.3: Possible orderings of concerns on \textit{SnowWhite}
### Problem identification

#### 3.2.4 Prince wants to join

The *Prince* wants to eat together with the dwarfs and *SnowWhite*. However reaching the forest where the dwarfs live takes him some time. We therefore decide to add an aspect to *SnowWhite* to make sure she waits with cooking until he arrives. The source code of these aspects is given in listings 3.7 and 3.8.

```java
public aspect Diet
{
  void around(SnowWhite sw): call (SnowWhite.cook(.)) && target(sw)
  {
    if (!House.isPresent(Prince))
    {
      wait();
    }
  }
}
```

Listing 3.6: *EatBeforeSix* aspect in AspectJ

```java
public aspect Diet
{
  void around(SnowWhite sw): call (SnowWhite.cook(.)) && target(sw)
  {
    // Not implemented in Java
    System.Scheduler.schedule(sw.cook(), 1800);
  }
}
```

Listing 3.5: *EatBeforeSix* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*

```java
concern WaitForPrince begin
  filtermodule syncprince begin
    conditions PrincePresent;
  end filtermodule
  superimposition begin
    selectors snowwhite = { SnowWhite };
    conditions snowwhite <= princepresent;
    filtermodules snowwhite <= syncprince;
  end superimposition
begin implementation in J#
  class PrinceChecker
  {
    public boolean princepresent()
    {
      return House.isPresent(Prince);
    }
  }
end implementation
```

Listing 3.7: *PrinceWantsToJoin* aspect in Compose*
3.3 Conclusion and scope

This section summarizes the previously mentioned types of conflicts and sketch the scope of this thesis.

Aspects are created to solve certain problems or to reflect the architectural aspects. The system requirements evolve over time and the system changes along with those requirements. This evolution may impact current aspects. New aspects can be created and old ones can be changed. The result could be that multiple aspects are applied to the same point. This can be caused by differences in timing. The aspects introduced due to changing requirements, are commonly defined at different times. The developers of the new aspects can differ from the original developers. Especially when systems become larger one aspect may impact a different part of the system without the new developer knowing this. As a result the combination of aspects may introduce conflicts. Aspects are often tested solely but not in combination with other aspects. The conflicts are mostly caused by lack of knowledge of the entire system architecture.

The definition of a conflict in this thesis is as follows:

*A conflict is a situation where a combination of filters reduce the functionality of themselves or of the system they are imposed upon.*

This definition is not specific to Composition Filters, it also holds for aspects in any other AOP
approach.

These conflicts tend to have several characteristics:

**Order of the filters is important.** If multiple aspects are imposed on the same object there can be numerous possible orderings. These orderings can be constrained, e.g. “filter A should be executed before all others”. However the conflict may occur only in one specific order. Imagine one filter executing a write on some property and subsequently another filter updating this property. This works fine in this order but if the order is reversed the write operation cancels the update operation. The conflicts are therefore asymmetric.

**Conflicts can occur when multiple aspects interfere with or overwrite one property.** Here the order can also influence the result; in any ordering there is a conflict. Imagine one aspect setting the value of some integer \( a \) to 2 and another aspect sets the value to 1. The resulting value of \( a \) depends on the chosen orders. But the conflicts still remains. This overwriting behaviour is one of the causes of conflicts.

**Another source for the conflicts is concurrency.** In concurrent systems there is often a need to synchronize between threads or to access the shared data. This should be done with monitors, critical sections and semaphores. Not using these techniques may introduce problems like: starvation, lifeless and deadlock. The overwriting behaviour gets even worse when aspects are accessing and writing the shared data. It should be noted that there is no concurrency in one filter module itself. All filters in one filter module are executed as one atomic action.

**One last cause of the conflicts is insufficient understanding or knowledge of the side effects of filters.** Most conflicts are not observed directly, they influence some property. Indirectly this can lead to conflicts that are not apparent just by looking at the filter specification. To detect these conflicts we need to capture this indirect property. However if this property is not clearly visible or made explicit in the filters the combination with other filters may seem harmless.

The next chapter will give a more detailed view of the origin of the conflicts and how they can arise due to composition. Subsequently the chapter will formulate a model which is capable of detecting the conflicts mentioned above. Chapter 5 will present an instance of this model for the current state of art.
In the previous chapter, the principles and some examples of the AOP approach has been explained. It should have been clear so far that the combination of aspects of join-points may yield conflicts. To develop a toolset component which is able to detect such conflicts and report them to the user, a clear description of the total AOP execution environment is necessary.

Therefore we start this chapter with definitions of the terms used in the AOP execution model. Using these terms the execution model will be explained and formally specified. This formal model utilizes the abstract concepts of actors, resources and operations on resources.

The resulting model has a very generic nature. Chapter five presents an instantiation of the model structure, with the values and conditions for the current set of composition filters and resources.
4.1 Filter execution model

Aspects can be introduced during the development of a software architecture, to allow for the evolution of the system requirements. These new aspects can be imposed on existing or new objects or aspects. Aspects can be crosscutting, if we want the control to be localized in one component. This component can be a separate component.

In the Composition Filters the aspects are called concerns. These can contain one or more filtermodules and a superimposition specification. The filtermodules contain filters, these are the functional specification of a filtermodule. The filtermodule allows for reuse as the superimposition imposes the filtermodule on the desired concerns or object. The execution model of the filters is illustrated by an example. This example was introduced in sections 3.2.3 and 3.2.4. There were two concerns presented. Each concern had a filtermodule. Each filtermodule had one filter. The filtermodules SetEatTime and SyncWithPrince were both superimposed on the cook method of the SnowWhite object. The combined situation is shown in figure 4.1.

![Figure 4.1: Diagram of the Snow White tale, with aspects](image)

Due to the fact that filtermodules can be superimposed at the same joinpoint, the shared joinpoint. The superimposition of filtermodules on such joinpoints appear in two ways:

**Statically:** This is the most widely used and implemented superimposition mechanism. All joinpoints are known during compile time. With static superimposition we are capable of source code weaving for instance. The superimposition for the entire system can be calculated statically. This is currently implemented in the Compose* project.

**Dynamically:** With the dynamic mechanism the superimposition depends on runtime conditions. In this case the superimposition can only be calculated if the constraints can be evaluated statically. If the constraints depend on some runtime property this can not be done anymore. A typical example could be: “depending on the load of a system impose a fallback mechanism”.

There are analysis tools in the Compose* project that reason about possible conflicts due to superimposition. This superimposition information is vital for these analysis tools. In the dynamic case we could analyse both options, superimpose and not superimpose, or we could take a specific case, e.g. superimpose, and analyse that superimposition.
First the analysis of the actual superimposition possibilities is done by SANE, the superimposition information is known at this point. There may still be a number of options available but the analysis tools will iterate over all possibilities.

At a shared joinpoint the ordering of the superimposition may be important, thus can be constrained. These constraints can be specified in two ways:

**Fixed:** Here all possible orderings can be calculated exactly. Imagine a constraint specifying: “filtermodule SetEatTime should be executed before SynWithPrince”. During execution example this will yield one ordering. But the constraints could also be: “filtermodule SetEatTime must be executed before any other filtermodule”. Imagine that another filtermodule A is added to SnowWhite.cook method. Again there is an ordering issue. There are 3! (= 8) possible orderings. However the SetEatTime must be executed before the others, resulting in only 2! (= 2) valid orderings. The ordering analysis is handled by FILTH.

**Conditional:** The constraints on the ordering can have certain runtime conditions. These conditions can only be evaluated at runtime. In this case FILTH will generate all possible orderings. An example of a conditional constraint could be: “If the prince is on his way first execute filtermodule SynWithPrince and then SetEatTime”.

Unless the ordering specification states constraints about the filter type the ordering inside a filtermodule remains unchanged. The generated list of filters is the flattened superimposition of filtermodules.

The following section will analyse the behaviour and execution model of a filter.

A filter in Composition Filters is a unit of execution. This is executed atomically, as is the case for filtermodules. Filters can be applied to all inbound and outbound calls of the superimposed object. There are currently six types of filters: Error, Dispatch, SUBSTITUTE, META, Wait and Realtime. The latter three were already used and briefly explained in chapter 3. Depending on the type of the filter the functionality of the filter is different.

The architecture of a filter can be decomposed into multiple filter elements. These filter elements have a condition, message matching part and substitution part. The condition part consists of a boolean expression. The message matching part has a matching pattern. The substitution part has a substitution pattern, this pattern is similar to the matching pattern.

The condition part is boolean expression. It can consists of conditions of filtermodule and boolean values True and False. These elements can be linked together with logical operators: And, Or and Not. The evaluation of the boolean expression, yields a value. The matching part is evaluated next if the value is true, otherwise it will not be evaluated.

The matching part specifies which messages are evaluated by the filter. This pattern is common to all filters as they can all be evaluated by incoming message and system decides whether to accept or reject. This is however specific for each filter instance, as the matching almost always
differ for each filter. Together with the condition part the state of conditions, the target and selector of the incoming message are checked. Imagine the following specification: \( \{ C_1 \Rightarrow [A.b]\} \), this reads all of the present information. However in the pattern: \( \{ True \Rightarrow [A.*]\} \) only the target is really checked. This forces us to instantiate this matching pattern for each filter.

The substitution part is used by the filters to explicitly change a property of a message. In the case of a dispatch filter: \( disp: Dispatch = \{ True \Rightarrow [A.\ast]B.\ast\} \), it explicitly changes the target to \( B \) for all the message matching the pattern. The substitution is also used for specifying temporary changes. An example is the meta filter where the substitution part specifies the object which handles the meta call, e.g. \( meta: Meta = \{ True \Rightarrow [A.\ast]logger.log\} \). This substitution part also depends on the filter instance. The exact changes are explicitly specified in this substitution part, which is optional. If there is no substitution part specified, the substitution part will be empty. It will not substitute any property. The substitution depends on the filter type. For some filter a substitution pattern may be irrelevant, in the case of an error filter. This is however a semantic constraint and we assume that it is handled by the parser.

A filter can do two things, either accept or reject. Which one is taken depends on the matching parts. A filter accepts if and only if the condition is true and the evaluation of the message processing yields a truth value for that message. Otherwise it will reject. This is the only component which is specific to a filter type.

Currently all filters have one more property on common. All of the filters have a “continue to the next filter” behaviour associated with either reject or accept. In a conflict detection tool we want to reuse the already present behaviour as much as possible. We therefore decide to make an abstraction and assign a number of actions to the decision.

These actions represent the behaviour of a filter when it accepts or rejects. In the case of a wait filter it would \textit{continue} to the next filter if it accepts. If the filter rejects it will \textit{wait} until it accepts. In the future as new filters are developed the developer of these new filters can reuse these behaviour. Chapter 5 will present an analysis of the current filters and their actions.

We will now show the execution of a filter in form of a flow chart. The chart is given in figure 4.2 and subsequently explained.

![Figure 4.2: Filter architecture](image-url)
The filter receives a message. The condition expression is evaluated and determines whether to continue to the message matching part or to reject. The message matching part has a part that does the actual matching of the message. If the current message matches the filter will accept otherwise it will reject. The substitution part can change certain properties of the message. Subsequently the actions are carried out by the filters, what action is performed depends on the above mentioned matchings and the filter type.

The substitution parts and the message matching parts are *generic* for every filter. We therefore decide to model these as separate actions. These actions are instantiated for every filter and are performed before and after each filter action. We now have the situation where we extracted the common behaviour into the message matching and substitution part. The accept and reject actions are specific for the filters. They represent the functional behaviour of the filters without the *generic* parts.

The specification of the Snow White filters is listed here:

```plaintext
1
settime : Realtime = { True => [ *.cook ] message.deadline=1800 ;
2
sync : Wait = { PrincePresent => [ *.cook ] ;
```

Listing 4.1: Diet aspect in Compose*

Table 4.1 shows what actions the example Snow White filters can have and under what circumstances.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Matching</th>
<th>Accept/Reject</th>
<th>Filter action</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>SnowWhite.cook</td>
<td>Accept</td>
<td>setrtprop</td>
</tr>
<tr>
<td>True</td>
<td>! (SnowWhite.cook)</td>
<td>Reject</td>
<td>continue</td>
</tr>
<tr>
<td>PrincePresent</td>
<td>SnowWhite.cook</td>
<td>Accept</td>
<td>continue</td>
</tr>
<tr>
<td>PrincePresent</td>
<td>! (SnowWhite.cook)</td>
<td>Reject</td>
<td>wait</td>
</tr>
<tr>
<td>! (PrincePresent)</td>
<td>! (SnowWhite.cook)</td>
<td>Reject</td>
<td>wait</td>
</tr>
</tbody>
</table>

Table 4.1: Filter action table for the Snow White filters

With these actions we can illustrate a filtermodule. We do this by presenting the filtermodule as a flow graph. The nodes of this flow graph are the filters and the edges are the actions. There are filters that can hold or interrupt the execution of the current filtermodule, these are currently *Error, Dispatch* and *Meta*. *Dispatch* can dispatch to another object. The error filter can hold the execution as a whole. The meta filter may or may not return to the filtermodule after carrying out its task. For the running example the graph is given on the left hand side of figure 4.3. If we add a dispatch between the realtime and wait filter, then the resulting graph will look like the right hand side of figure 4.3. The condition, matching and substitution parts have been omitted for clarification purposes.

Filters do not only interact with the conflicting areas. Certain conflicts may become apparent when there is actually some things done with the conflicting property. A violation of the realtime constraints is often really a conflict if there is a realtime *scheduler* present. Then the combination with the wait filter may render the scheduler useless. Another example could be changing the state of a condition. Changing the condition is on itself not a conflict. However if the *system* object on which is superimposed depends on this condition, there could be a problem.
The system and scheduler in the mentioned examples are so-called external entities. They are not present in the filters but in the execution environment of the filters.

This section analysed of the filter execution environment. This environment is the basis for the conflict detection approach presented in the next section.

### 4.2 Conflict detection approach

The previous section described the execution model of the filters. Those filters were translated into actions depending on the accept or reject behaviour.

The actions are however not sufficient for the detection of the conflicts. There are a number of reasons why the detection based on the actions will not be very successful:

**The actions cover a large functional area.** We want the detection of conflicts to be as precise as possible. Therefore we need more detail than just one action. However the actions do provide us with reusability, as more details would mean less reusability.

**The conflicts also tend to happen in the side effects of the actions.** These side effects are not easily traced back to one specific action.

So we need to create a layer which abstracts the details of the conflicts and permits reusability of the actions. We create an Abstract Virtual machine which make the details explicit. It offers an interface for the actions of the filters and of the external entities. This allows us to be specific with the detection of conflicts and at the same time providing us with the reusability for the actions. The resulting situation is depicted in figure 4.4.

As stated previously conflicts are often not captured by one specific system property. We need to capture the area where they do conflict. Such conflicting area can show the place where side effects occur. This conflicting area is represented by a resource.
A resource represents:

- A concrete property of the system or filter. For example the target of a message.
- An abstract property that represents the area where two filters have side effects. It could also represent a non-existing property but which encapsulates the conflict area best. For instance a timing resource which represents the notion of time.
- A composed property, a conflict can sometimes occur on a combination of properties. For example state, this encapsulates the system state and the conditions of a message.

We now need to make a mapping between the actions and the resources. The actions act on a virtual machine, they are subsequently translated as operations on resources. An operation is an atomic unit of execution inside the virtual machine.

For example an operation can be: read, write, increment, reset. These operations are completely filter programmer adaptable and expandable. The virtual machine is shown in figure 4.5.

If we look at the running example we can create a table where the actions and matching parts are translated to operations. Table 4.2 presents this. The table is now explained in more detail.
The “settime” matching part does not read the *state* as the used condition is true. The *target* is not read by either “settime” or “sync” matching parts, the *selector* however is. The matching part of the “sync” filter reads the condition *PrincePresent*, therefore it reads the state. Both filters do not have a substitution part, so these are not present in the table. The “setrtprop” action of the “settime” filter sets the *deadline* resource. The “wait” action of the “sync” filter write the *timing* resource. The “continue” actions for both filter do nothing.

The previous chapter introduced the notion of external entities that could also act on the virtual machine and therefore also operate on the resources. The entities *System* and *Scheduler* were mentioned. These external entities are also added, the result is shown in table 4.3.

<table>
<thead>
<tr>
<th>Action</th>
<th>target</th>
<th>selector</th>
<th>state</th>
<th>timing</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>settime.condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>settime.matching</td>
<td></td>
<td>read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>settime.setrtprop</td>
<td></td>
<td></td>
<td></td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>settime.continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>settime.substitution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sync.condition</td>
<td></td>
<td>read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sync.matching</td>
<td></td>
<td>read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sync.wait</td>
<td></td>
<td></td>
<td></td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>sync.continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sync.substitution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: External entities added to the table.

All the filters are evaluated in the order in which they are presented. For each filter a number of actions is carried out on the virtual machine. These actions are translated as operations on resources. Their order is also retained here. When all filters are evaluated, the effected resources have a sequence of operations. The operation sequences for the Snow White example can easily be extracted from table 4.3. They are shown here:
4.3 Formal Specification of the model

Given these operation sequences we need to define a conflict in this sense. One could create a table where the operations are checked against the other operations. However when conflicts become more complex a table is not well suited. An example of this complexity would be the following: the operation sequence “read write read” is non-conflicting but the sequence “read write read write” is. This is hard to express in a table. Therefore we chose to specify a regular expression for the non-conflicting sequence. The regular expression allows us to easily specify difficult sequences.

A resource is conflict free when the operation sequence on that resource matches the regular expression. This analysis against the regular expression is done for every affected resource.

The model presented in this section does not consider under what conditions and which matching part do the filters accept or reject. As a consequence a conflict may be detected which is not present as the execution of the filters can never occur in that manner. These kinds of conflicts are filtered from the output as they are not of interest.

The filters are declarative. This allows us to specify exactly under which conditions and which message could trigger this conflict. This could be extremely helpful in debugging code and tracing the application. Both this specialization and filtering of non conflicts is done with the aid of FIRE. FIRE can given the accept and reject behaviour check of the conditions and messages are evaluated by the filters. If this set is empty the order may never occur and the conflict is also non existing.

This model is derived from the Bernstein [4] conditions introduced in 1966. These conditions state that depending on read and write operations of processes on resources, certain concurrency access conflicts can be identified. Bernstein based his model on the division of process actions in sets of readers and writers on resources.

In this environment of analysis of concerns, the order in which the actions appear is crucial. The conflicts are asymmetric in their nature; they are order dependent. If the conflicts were symmetric a set of actions would have been sufficient. By using sequences of actions on resources instead of sets we are able to include the order of the actions, then our model can be thus seen as an extension to the Bernstein model.

This model is sufficiently generic to host all currently known filters and actions. In addition it should be able to handle future extensions.

4.3 Formal Specification of the model

The model is formally specified in VDM. VDM stands for the “Vienna Development Method”. VDM-SL is a specification language which was developed originally in the beginning of the 70’s at IBM in Vienna. The VDM-SL notation became an ISO standard in December 1996. In 1995 an object oriented extension was made, called VDM++ [10] and [11].
The next section will introduce some basic data types. Then the internal data structures used in the model are defined and explained. Subsequently the operations on these data structure are presented.

### 4.3.1 Type Definitions

First we define the types used throughout this specification. Resources and the Actors (or processes) are identified by their names.

\[
Name = \text{char}^+ \\
\text{Resourcename, Actorname} = Name \\
\text{Alphabet} = \text{Name-set} \\
\text{ActionType} = \text{Alphabet} \\
\text{Regex} = \text{is not defined here}
\]

A Name is a sequence of at least one character, the resource and actor names are of this type name. The possible actions on a resource are specified strings of an alphabet. In this chapter we use the two operations read and write as example, represented by the alphabet \{r,w\}. Anticipating future extensions we have left the possibility open to use additional or other symbols. Other actions might be e.g.: increment, decrement, exchange, set zero.

Any sequence of actions, is represented by a sequence of ActionType. Such a sequence can be considered as an instance of a regular expression (a pattern) built up from the alphabet symbols and the operators (\(*,|,+,*\)). Because the theory on regular expressions is so mature[34] it is not specified here.

A set of type Resources is created which is a set of known resource-names. Then an ActionDescription record is defined, this is a tuple with an Actorname and an ActionType. This specifies which actor does what action on a resource.

\[
\text{Resources} = \text{Resourcename-set} \\
\text{compose ActionDescription of} \\
\text{actor} : \text{Actorname}, \\
\text{action} : \text{Actiontype} \\
\text{end} \\
\text{compose ResourceDescription of} \\
\text{alphabet} : \text{Alphabet}, \\
\text{checkregex} : \text{Regex} \\
\text{end}
\]

A Resource has two attributes in its description: the names of the possible operations (its alphabet) and a regular expression built up from these characters and the operators. If the actual sequence of actions on this resource matches its regular expression, the actions for this resource are considered conflict free.
4.3.2 Internal data structures

The different high level structures used to store the information of the readers and writers on the resources are now defined using the given types and sets.

The persistent storage structure holding the attributes for the known resources at any moment is a map. This ResourceDescriptionMap (RDM) maps ResourceName to the ResourceDescriptions with their actual values. The persistence can be implemented for instance by loading the contents of the map from a file in permanent storage during initialisation of the tool.

\[ RDM = \text{ResourceName} \mapsto \text{ResourceDescription}^* \]

where

\[ \text{inv-RDM} \triangleq \text{dom RDM} = \text{Resources} \]

The invariant for the RDM formulates the requirement, that for each known Resource a ResourceDescription is defined. The domain of the map is the same as the set Resources. A graphical representation of the map and the methods is presented in picture 4.6.

![Figure 4.6: Resource Description Map](image)

With the above described entities the Resource Usage Map, abbreviated as RUM, is defined:

\[ RUM = \text{ResourceName} \mapsto \text{ActionDescription}^* \]

where

\[ \text{inv-RUM} \triangleq \text{dom RUM} \subseteq \text{Resources} \]

This map states that a RUM is a general map from resource name(s) to a sequence of ActionDescriptions. Here the invariant states that the map domain is must be a subset of the Resources. The checking algorithm can easily traverse the map left-hand side and check whether a conflict exists for this resource. Again a graphical representation of the map and the methods is presented in picture 4.7.
4.3.3 Model operations

Now that the general data structures are defined we can add operations on these elements. This completes the model. Three operations are specified: \textit{AddResourceDescription}, \textit{AddActionDescription} and \textit{CheckResourceConflict}.

The first operation allows the introduction of a new resource to the set of known resources and defines how the attributes for this resource are stored.

\begin{equation}
\text{AddResourceDescription } \left( rsrc: \text{ResourceName}, \ alf: \text{Alphabet}, \ regex: \text{Regex} \right)
\end{equation}

\begin{itemize}
\item \text{ext wr } RDM \colon \text{ResourceName } \xrightarrow{m} \text{ResourceDescription}^* \\
\item \text{post } \text{Resources} \cup \{rsrc\} \land \\
RDM [rsrc] = \text{mk-} (\ alf, \ regex)
\end{itemize}

The postcondition states that the set of known resources is extended with the new \textit{ResourceName}, a \textit{ResourceDescription} record has been allocated and added to the map. It explicitly states that the \textit{RDM} will be changed by this operation via the \text{ext wr} property.

Imagine we want to introduce a \textit{ResourceDescription} for \textit{ResourceNameXX}, with \textit{alphabet} \{r,w\} and \textit{regex} for non conflicting paths: \textit{r*w*}. This \textit{Resource} and its \textit{ResourceDescription} is added by invoking the operation:

\begin{equation}
\text{AddResourceDescription } \left( \text{ResourceNameXX}, \ \{r,w\}, \ r^* w^* \right)
\end{equation}

The second operation processes the dependency \textit{tuples} delivered by a preprocessor and stores them properly in the \textit{ResourceUsageMap}.

\begin{equation}
\text{AddActionDescription } \left( rsrc: \text{ResourceName}, \ ad: \text{ActionDescription} \right)
\end{equation}

\begin{itemize}
\item \text{ext wr } RUM \colon \text{ResourceName } \xrightarrow{m} \text{ActionDescription}^* \\
\item \text{rd } RDM \colon \text{ResourceName } \xrightarrow{m} \text{ResourceDescription}^* \\
\item \text{pre } \text{rsr}c \in \text{dom } RDM \\
\item \text{post } RUM [rsrc] = RUM [rsrc] \cup add
\end{itemize}

The operation adds a new action to a resource in the map. The precondition represents the
requirement that the value of parameter \( rsrc \) is an element of the set of known resources, the
domain of the RDM, for which a \( ResourceDescription \) has been defined. The post condition
of this method states that the new \( RUM \) entry for this resource is equal to old \( RUM \) entry
(indicated with the arrow on top of it) concatenated with the new \( ActionDescription \).

Referring again to the well developed theory of regular expression[34], we introduce here the
function \( regexCheck \). It takes a sequence of actiontype characters and a regular expression as
input and delivers a boolean true value if the pattern matches the expression. The check will
yield a boolean false value which means a match between the pattern and the input can not be
made.

\[
regexCheck : actiontype^* \times \text{Regex} \rightarrow \mathbb{B}
\]

\[
regexCheck(\text{actionseq}, \text{regex}) \triangleq \text{externally defined}
\]

With the regular expression for a resource as the input the model checks this expression against
the generated stream of actions. If the expression does not match we have detected a conflict.
When all action-descriptions are processed, the tool checks whether there are conflicts on the
basis of the contents of the model data structures. As discussed previously the conflict or
dependency detection requirements are entirely user adaptable.

The checking operation for a specific \( ResourceName \) can be specified as:

\[
\text{CheckResourceConflict} (r: ResourceName) b: \mathbb{B}
\]

\[
\text{ext rd RUM} : Resource \xrightarrow{m} ActionDescription^*
\]

\[
\text{rd RDM} : Resource \xrightarrow{m} ResourceDescription^*
\]

\[
\text{pre r} \in \text{Resources}
\]

\[
\text{post let actionseq: actiontype}^* \text{ in}
\]

\[
b = \forall x: [0..\text{len}(\text{RUM}(r))] \cdot \text{actionseq}[x] = \text{RUM}(r)[x].actiontype \land
\]

\[
\text{regexCheck(\text{actionseq}, RDM(r).checkregex)}
\]

In the post condition we construct a sequence of alphabet characters \( actionseq \) from the action
descriptor sequence. Then a \textbf{true} verdict is delivered if \textbf{and only if} the \( actionseq \) matches the
\textit{checkregex} pattern that was given in the corresponding ResourceDescription. If this holds for
each resource-name the analysed filters are conflict free.
The model presented in the previous chapter is generic and should be able to handle all currently known filters and their actions.

This chapter will discuss the step in between the generic model and the implementation in the toolset. The model data-structures need to be “filled” with the information specific to Composition Filters, for a given set of filters, and for a specific conflict model. This intermediate step can be seen as the “instantiation” of the model.

First we will analyze the current filter set and subsequently identify and analyze the external entities. This is followed by a section of the used conflict model.
5.1 Filter analysis

To detect which resources, actions and operations are necessary an analysis of the current filters is needed. Future extensions to the set of filters can be analysed in a similar manner.

This section will describe for each filter:

- the functionality and its use,
- the accept and reject actions (if applicable),
- the resources used by the filter behaviour,
- the operations performed on these resources by the filter actions.

5.1.1 Condition

Description
The architecture of a filter can be decomposed into multiple filter elements. These filter elements have a condition and a message matching part. The condition part consists of a boolean expression. The condition part is common to all filters.

Used resources
The matching part as presented here has two main elements; the condition and the message pattern. This matching part will never change sometime it only reads certain properties. An example of such a message part could be: \( \text{"PrincePresent} \Rightarrow \{\text{SnowWhite.cook}\} \). It reads the condition PrincePresent. The state is more abstract, it represents entire state of the system and of the message. The filter can use conditions which depend on some state. As stated in chapter 4 these all fall under the state.

Performed operations
The operations performed by all the filters on the above described resources depend on the filter instance. The operations used in this model instantiation are however limited to two possible operations:

- **read(r)** : A read operation is interpreted as accessing that resource without changing its state.
- **write(w)** : The write operation is interpreted as changing the state of the resource.

The operations, read and write, are directly derived from the conditions of Bernstein [4]. Other possible operations could be: update, increment, decrement, set or reset. This is filter programmer adaptable. We chose the read and write actions as these serve the purpose of the current filters. In a future study new operations might be introduced.

The alphabet resulting from the operations on the resources is: r,w. As stated the matching part is specific for every instance of a filter. Imagine the following matching part: \( \text{"True} \Rightarrow \{\ast.cook\} \) which is equal to \( \{\text{cook}\} \). As one can see the filter part does use a condition however this condition is always true. It therefore does not read the state.
5.1.2 Matching

Description
The architecture of a filter can be decomposed into multiple filter elements. These filter elements have a condition and a message matching part. The message matching part has a matching pattern. The matching part specifies which messages are evaluated by the filter. This pattern is common to all filters as they can all evaluate the incoming message to decide whether to accept or reject.

Used resources
As stated in 5.1.1 the matching part will never change sometime it only reads certain properties. The same example is taken: "PrincePresent ⇒ [SnowWhite.cook]". It uses the target, in this case SnowWhite, and selector, cook, of a message. The target is the current destination of the message, this can be the imposed object or the concern itself. The selector is the current method name.

Performed operations
As stated the matching part is specific for every instance of a filter. Imagine the following matching part: "True ⇒ [SnowWhite].cook" which is equal to "[cook]". This matching part does not read the target. The possible targets for the object are limited by the superimposition specification. In this case the matching part only reads the selector. One can imagine that other combinations are also possible.

5.1.3 Substitution

Description
The substitution part is used by the filters to explicitly change a property of a message, e.g. target or selector. The substitute part depends on the filter type, filter instance and matching part. To explain this consider the following matching part: "True ⇒ [SnowWhite].Prince\#." This substitute states that every call to SnowWhite should be evaluated and the target of the message should be changed to the Prince. The substitution is instantiated for every filter, and the specific properties those are substituted should be added as resources.

Used resources
The substitution part comes in many different flavours. The exact used resource depend on the specification of the substitution. The most widely used form of substitution is to change the target or selector.

Performed operations
The substitution part changes certain resources, all affected resources are therefore written.

Now the specification of every filter is discussed separately.

5.1.4 Error filter

Description
The error filter is typically used for checking constraints. These can be pre- or postconditions
of the message or constraints on the system state.

**Behaviour**
The error filter has two associated actions. If the filter accepts, the message continues to the next filter, this is the *continue* action. If the filter rejects an (user-defined) exception is raised, this is the *error* action.

**Used resources**
The error filter itself does not operate on other resources then the matching part. We could model the error filter as a dispatch filter. The filter can specify a custom exception. For each error filter this should be checked which exception should be thrown. This can be interpreted as a dispatch to an exception handler. In this case it would write the *target* and *selector*. This is however not modelled here.

**Performed operations**
The condition and matching part of this filter reads the *target*, *selector* and system *state*. This actions of these filters do not operate any resources. One could imagine that maybe in the future an exception could be modelled as a resource and that this error filter writes this exception. At the time of writing we do not feel that there is a need for this resource. The filter may also have a substitution part, this is however ignored in the error filter.

### 5.1.5 Dispatch filter

**Description**
The dispatch filter is primarily used for delegation purposes and dynamic inheritance. If the matching pattern matches the given signature the substitution part will be carried out. After the substitution has been carried out the message will be dispatched to the delegate.

**Behaviour**
If the matching part of the dispatch filter matches, the message will be dispatched to the delegate, this is considered to be the *dispatch* action. If it does not match the filter expression, the message continues to the next filter, this is the before mentioned *continue* action.

**Used resources**
The matching part has been covered in section [5.1.2](#). The substitution part related to this message can change the *target* and *selector*.

**Performed operations**
The substitution part of the filter writes the *target* and *selector*. The matching part of this filter is given in section [5.1.2](#).

### 5.1.6 Meta filter

**Description**
The meta filter is the most powerful filter currently available. If the filter matches, the current message is reified and sent as a parameter to a (meta) object. This object can manipulate the message and may reactivate its execution. It may be the most powerful filter, however can make
reasoning about the meta filter more difficult than other filters. The meta action may change the entire system behaviour.

**Behaviour**
If the matching pattern of the meta filter accepts the meta action is executed. Subsequently the message will be reified and sent to the meta object specified in the substitution part of the meta filter. If the matching part of the filter rejects the message continues to the next filter.

**Used resources**
The meta filter can may change every property of the system and message, it may also change the sender, server and parameters of a message. The sender represents the object which initiated the message. It can be used to identify the originator of a message. The server is the dynamically bound self or this reference, as the object moves through a dispatch chain this will change with every call, where the sender may remain the same. The parameters are the arguments of the message currently under investigation.

**Performed operations**
If the matching parts accepts the substitution is carried out. This substitution temporary changes the target and selector to the meta object. This change does however not hold for the current message as this is reified and passed to this object. The substitution part does not change the target and selector. The meta action itself can however change these properties and more properties of a message, like parameters. In the future as a possible meta specification of the meta filters are going to be introduced, the affected resources and the operations on these resources can exactly be pinpointed.

5.1.7 Substitute filter

**Description**
The substitute filter can be used to change the properties of the message. Imagine a change of the selector so that the message is accepted by the underlying object.

**Behaviour**
If the matching part of the filter matches, it will carry out the substitution specified in the substitution part and continues to the next filter. If it does not match, the continue action, the message continues to the next filter.

**Used resources**
The substitution part of the substitute filter can explicitly change properties of the message. The most changed properties are target and selector. These can however be any property of a message, like sender, server and parameters.

**Performed operations**
The matching part is already discussed. The substitution part specifies the explicit changes, writes, on the resources.
5.1.8 Wait filter

Description
The wait filter is used for synchronization constraints. If the matching part does not match the filter will wait until the matching part matches. If a message is received which does not match the matching part the message will wait indefinitely.

Behaviour
If the matching part does not match the filter will wait for some time. If it matches the message continues to the next filter.

Used resources
If the wait filter rejects it will operate on the timing, this resources represents the notion of time in the system. It is used when dealing with real-time systems where timing is crucial for the scheduler.

Performed operations
The reject action performs a write on the timing property. The matching part is already discussed. The wait filter has no substitution part.

5.1.9 Realtime filter

Description
This filter is used to set certain real-time properties of a message. These can be starttime or endtime. For this thesis we only look at the message deadline property, further research may be needed to add more detail here.

Behaviour
If the matching part of the realtime filter accepts, the realtime properties are adjusted. This is the setrtprop action of the filter. If the matching part does not match, the message continues to the next filter.

Used resources
As stated earlier it changes the deadline of the message. In either case, accept or reject, it will also operate on the target, selector, state.

Performed operations
The setrtprop action, the accept behaviour, writes the deadline.

5.2 External entities

5.2.1 System

Description
The system represent the underlying system. This is abstract entity as it can be virtually anything. It can be the target of the filtered message, either the imposed object or the imple-
5.3 Resource usage overview

mentation. This target may want to read the parameters of the message. It can also change the state of the system. The assumption here is that certain conflicts occur only when something actually is using the changed information. This is modelled as system actions.

**Behaviour**
As the system is not a filter which has an accept or reject action, it is an action by itself.

**Used resources**
The system uses the following resources: selector, target, state, server, sender, parameters and timing.

**Performed operations**
As the system may be constantly changing this may be subject to the most changes, in the current state however we stipulate the following operations. The system reads the selector, target, server, sender and parameters and writes the timing and state.

5.2.2 Scheduler

**Description**
The scheduler represents the scheduler in a realtime or multithreading system. Its job is to schedule every message in such a way that the deadline of each message is not violated. If it cannot find a way to schedule a message in the correct way, it will raise an exception.

**Behaviour**
As with the system the scheduler does not have the accept or reject behaviour.

**Used resources**
The scheduler needs the target and selector in order to determine the message. It also uses the deadline and timing to check whether the deadline or this message is still valid.

**Performed operations**
The scheduler read all the above mentioned resources.

5.3 Resource usage overview

Table 5.1 summarizes the presented filters, actions, operations and external entities:

The table shows that the behaviour of the current filters is almost solely in the condition, matching and substitution part. Only the realtime, wait and meta filter have a behaviour outside the condition, matching and substitution part. As we do not know what kind of filters can be created in the future we still feel comfortable with the action model. Even though certain explicit properties, like the deadline in the setstrprop action, can also be put in the substitution part. The optional read at the condition part states that it depends on the boolean expression of this condition. The possible read for the matching actions are determined by the matching patterns. If the writes for the substitution will be done depends on the substitution expression.
The state property now encapsulates the whole system state and the state of the conditions. For the conditions in the filters this can be expanded to all the exact used conditions. The filters are declarative, therefore we can get this information easily from the filter specification. We can then instantiate the condition part for each filter, with the actual conditions that are read. The same holds for the matching and substitution parts. These can be instantiated for each filter with the specification. This instantiation will create dynamically extra columns. If a new filter type is added this table can be extended to add the actions and resources. If the new filter type only introduces a new action we can simply add a row in the table.

### 5.4 Unified Filter

The previous sections described the current filters. It also concluded that most of the functionality is currently present in the substitution part. This section presents an abstract filter based on these conclusion. This filter is created with respect to the semantic reasoning.

The unified filter has just like the current filter a condition and matching part. It also has an accept and a reject behaviour.

\[
\text{filter} : \{ \text{condition expression} \land \text{matching part}(s) \Rightarrow \\
\{ \text{accept substitution}(s) \} , \{ \text{reject substitution}(s) \} \}
\]

All the current filters can be expressed in this filter. The used resources are specified directly in the filter. The following list will evaluate the section and explain which operations can be performed on the resource.

**condition part** : As discussed this reads the state, this can of course be expanded to concrete conditions in the resources.

**matching part** : This part can read the target and selector of a message.
substitution part: The substitution part can write recourses. These resources can be any
the the filter can possibly change.

All filter specific information is in the substitution part. The substitutions that are carried out
can then be considered write operations on resources. A realtime filter could then be specified as:
realtime: \{ True \land \ast . cook \Rightarrow \{} , \{ message . deadline = 1800 \} \}. The wait filter in the Snow White
example is then specified as: wait: \{ PrincePresent \land \ast . cook \Rightarrow \{} , \{ timing \} \}. A dispatch filter
can also be specified in the same sense: dispatch: \{ True \land \ast . cook \Rightarrow \{} , \{ message . target = Doc \} \}.

5.5 Conflict model

The definition of a conflict has been a point of much discussion, therefore a parametrized form
of this definition was defined. This definition is presented as a table and translated to a regular
expression. The model just checks which sequences of actions are valid by matching them to
the regular expression. One example of such a dependency table is given in table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>∨</td>
<td>×</td>
</tr>
<tr>
<td>write</td>
<td>×</td>
<td>∨</td>
</tr>
</tbody>
</table>

Table 5.2: Possible dependency table

The table should be interpreted as follows. If a resource is read (row read) then it may be
read again (column entry) or written. If however a resource has been written it may not be
read anymore, it may only be written. As stated in chapter 4 a table is only useful when the
non conflicting patterns are easy. However if these patterns are more complex a table is not
sufficient. We use a regular expression to express these complex patterns. From the table we
can derive the non-conflicting regular expression: r* w*. An example of an operation sequence
could be: r r r w r.

The given example sequence does not match the regular expression, as in the sequence the
last r is not allowed. Also for other types of resources, read-only(r*), write-only(w*) and
everything((r*w*)r) could be given as non-conflicting action sequences.

The definition of a conflict has, in general, the tendency to be commutative, meaning ordering
does not matter, also known as symmetric. Whereas in this case order does matter, a read
followed by a write is different than a write followed by a read. A dependency is a more
appropriate word here as the problems are asymmetric.

The filters are defined into four parts; condition, matching, substitution and action part. Con-
flicts that are detected between these parts are considered to be non conflicts. As this would
mean that there is a conflict inside the filter. The filters are however defined as non conflicting
within the filter itself. We therefore discard the possible conflicts between the different parts of
a filter.
5.6 Instantiation of the problems

5.6.1 Keeping Snow White healthy

In sections 3.2.1 and 3.2.2 I introduced an example of a conflict that I would like to be detected. The listing 5.1 shows the two filters that were conflicting.

<table>
<thead>
<tr>
<th>resource</th>
<th>action descriptions</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>selector</td>
<td>CheckFood.check.matching</td>
<td>⇒ r</td>
</tr>
<tr>
<td></td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.matching</td>
<td>⇒ r</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>server</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>sender</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>parameters</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>timing</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td>state</td>
<td>CheckFood.check.meta</td>
<td>⇒ w</td>
</tr>
<tr>
<td></td>
<td>DietCheck.check.meta</td>
<td>⇒ w</td>
</tr>
</tbody>
</table>

Table 5.3: Snow White RUM after evaluating the filters

Adding external entities

We introduced the notion of external entities in section 4.1. After adding the System and Scheduler actions on the virtual machine the ResourceUsageMap is shown in table 5.4.

At this point we can start to analyze where the conflicts occur. For this analysis we use the in section 5.5 presented regular expression: $r^* w^*$. As one can see from the operation sequences there a lot of conflicts present. If we take a closer look at the selector we can see that the
### Table 5.4: Snow White RUM after evaluating the filters and external entities

<table>
<thead>
<tr>
<th>resource</th>
<th>action descriptions</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ r &lt;br&gt; Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>selector</td>
<td>CheckFood.check.matching ⇒ r &lt;br&gt; CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.matching ⇒ r &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ r &lt;br&gt; Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>sender</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>deadline</td>
<td>System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>timing</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ w &lt;br&gt; Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>CheckFood.check.meta ⇒ w &lt;br&gt; DietCheck.check.meta ⇒ w &lt;br&gt; System ⇒ w</td>
<td></td>
</tr>
</tbody>
</table>
operation sequence is: \textit{rwrwrr}. This does no match the given regular expression. The first two operations are however on the same filter. We stated earlier that a conflict in a filter is not conflict. The same hold for the next two operations. We can rewrite the operation sequence to: \textit{(rw)rwr}. Everything between parenthesis is considered to be alright. However even then there is still a conflict. The last two operations are reads and the previous operations did contains a write operation. It therefore does not match the regular expression.

However the conflict we were after is not present here, there are conflicts but these are not on the area where we expect them to be. We were expecting a conflict on the state as this is where the disjoint conditions were encapsulated. In stead there is an operation sequence: \textit{www}, which does match the regular expression. In chapter 7 I will present a possible solution for this problem using the Object Constraint Language.

In the implementation this entire process is automatic. The generated conflict report is presented in section 6.4.1.

### 5.6.2 Diner in time problem detected

As illustrated in section 3.2.3, there are several possible orderings of the \textit{Diner in time problem}. For this section we will only show one order. The same analysis can be carried out for another ordering. In the implementation chapter both orders will be automatically checked and for each order a conflict report generated.

The order we use in this example is:

```
1 EatBeforeSix.check : Realtime = { True => [ * . cook ] message.deadline=1800 }; 
2 PrinceWantsToEat.sync : Wait { PrincePresent => [ * . cook ] }; 
```

Listing 5.2: \textit{Diner in time} Compose* concern

The evaluation of the filters is sequential, for every step the filter makes an entry in the \textit{ResourceUsageMap}. We assume that the \textit{ResourceDescriptionMap} is already filled with the appropriate information.

**Evaluating the filters**

All the filters are now evaluated, first their condition, matching and substitution parts are evaluated. After this the specific filter action are checked. The realtime filter has two actions, see section 5.1.9, \textit{continue} or \textit{setrtprop}. The \textit{setrtprop} action writes the \textit{deadline}. The wait filter can write the \textit{timing}. So after the evaluation of the realtime and wait filter the \textit{ResourceUsageMap} will look like table 5.5

<table>
<thead>
<tr>
<th>resource</th>
<th>action descriptions</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>selector</td>
<td>EatBeforeSix.check.matching ( \Rightarrow r )</td>
<td>( r )</td>
</tr>
<tr>
<td>deadline</td>
<td>PrinceWantsToEat.sync.matching ( \Rightarrow r )</td>
<td>( r )</td>
</tr>
<tr>
<td>timing</td>
<td>EatBeforeSix.check.setrtprop ( \Rightarrow w )</td>
<td>( w )</td>
</tr>
<tr>
<td>timing</td>
<td>PrinceWantsToEat.sync.wait ( \Rightarrow w )</td>
<td>( w )</td>
</tr>
</tbody>
</table>

Table 5.5: Diner in time RUM after evaluating the filters
5.6 Adding external entities

In this section we again add the external entities System and Scheduler. This result is shown in table 5.6.

<table>
<thead>
<tr>
<th>resource</th>
<th>action descriptions</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>System ⇒ r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>selector</td>
<td>EatBeforeSix.check.matching ⇒ r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrinceWantsToEat.sync.matching</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System ⇒ r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>deadline</td>
<td>EatBeforeSix.check.serrtprop ⇒ w</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>timing</td>
<td>PrinceWantsToEat.sync.wait ⇒ w</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System ⇒ w</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduler ⇒ r</td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>System ⇒ w</td>
<td></td>
</tr>
<tr>
<td>sender</td>
<td>System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>System ⇒ r</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>System ⇒ w</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Diner in time run after evaluating the filters and external entities

The operation sequence on the deadline resource is: wr. As the deadline of a message may change because we use realtime filters we do not consider a write followed by a read a problem. We therefore specify the non conflicting regular expression as: \((r^* w^*)^*\). The operation sequence for the timing is: wwr. There is a conflict detected here as it does not match the regular expression for this resource: \(r^* w^*\). The conflict on the timing is detected correctly. In the implementation this entire process is automatic. The generated conflict report is presented in section 6.4.2.
This chapter describes the effort to implement the described model. As the model was formally specified, the implementation of the core system itself was relatively straightforward.

This chapter is structured as follows, first the position in the whole Compose* toolset is explained, with respect to preprocessing and input. Subsequently the basic system is given. This is almost a direct mapping of the formal specification, as it should be. The input, output parsers and generators are briefly discussed, these do not enhance the functionality but try to offer a user friendly way of specifying input and generating output. The control flow of the SECRET tool is given as a sequence diagram to illustrate the interaction between the described classes. Then the two problems, introduced in chapter 3, are given as input for the tool and the results are shown.
6.1 Location in the Compose* project

As discussed in the second chapter SECRET is a small component in the whole Compose* toolset. The input for the model is largely dependant on the output of the tools working before SECRET. This section will highlight the necessary steps taken obtain the desired input for the tool.

The entire processing part can be divided into two main steps. In the first step the given concern specification is parsed and the relevant information is extracted. This information is then added to the central Repository in a predefined format. Then a tool called REXREF will go over all the parsed concerns and resolve any dependencies and references between those concerns. This tool is part of the so-called repository manipulators. Other examples of such tools are SANE, which reasons about superimposition and identifies the areas where multiple pieces of concerns are added to the same point. Then the FILTH tool reasons about the possible ordering of those filter modules.

After all these preprocessing steps have been carried out, the analysis tools can start with their work. For instance CORE can start checking for consistency problems in the given filters. This is also where SECRET comes in. For every possible ordering generated by FILTH it will analyse that ordering, and report conflicts. An important tool in for the analysis tools is FIRE, it provides an intelligent interface on the Repository. SECRET uses this interface to reduce the number of reported conflicts as the model may report conflicts that are not possible.

The picture 6.1 shows the placement with respect to the other tools, the parser has been omitted from this picture.

![Diagram showing the placement of SECRET](image)

Figure 6.1: Diagram showing the placement of SECRET

6.2 Implementation description

This section will present the implementation effort, it is divided into three main parts:

**Base system:** This is one to one mapping of the formal model.
XML parsers: As the filter specification is in XML, the tool needs to parse the file. This is done in this section of the tool.

Conflict reporters: This part deals with generating an error report for SECRET, e.g. in HTML.

The following sections explain the above stated parts in more detail.

6.2.1 Base system

As stated in [20] the use of formal methods can, when used properly, greatly reduce the coding, testing and debugging effort. This was also the case for SECRET, the core system was relatively easy to implement because most of the data structures used in the model were already present in Java [26] or were created with minimal effort. In figure 6.2 the UML class diagram is shown.

![UML class diagram of the core system](image)

The components shown in the class diagram are:

SECRET: This is the main entry point for the entire tool. The MASTER calls SECRET via its run method, a CommonResources object is passed as a parameter to setup properties like: debugging, path of the input XML file and location of the output files.

SecretRepository: This class is used to store all the parsed input filter specification. It also provides an intelligent interface on the stored data to allow easy access to certain types of information, e.g. getAllActionsForFilter.

AbstractVM: This is the partial embodiment of the formal model. All the operations and data structures presented in the model are present in this class, e.g. it has the ResourceDescrip-
The class diagram of the XML parser is shown in figure 6.3 and is explained.

The main part here is the SecretXMLParser, it is used to open the filter specification file and to parse it. It uses the standard SAX parsers provide by Java. It provides the detection of SAX events like start and end of tags.

The input specification consists of three main file.

The first file contains a “filters” section where every filter type is mapped to an accept and reject action. In the second file the mapping from these actions to operations on resources are stated within the resource actions section. Finally the resources themselves are provided with the information needed to detect conflicts on those resources. Each of these files have their own parsers. These are clearly shown in the UML diagram.

The above mentioned XML files are shown in the following listings.

```xml
<filters>
```
6.3 Control flow

This section will discuss the control flow in SECRET, with the aid of a UML sequence diagram. This diagram is shown in figure [6.4].

The MASTER is the main entry point and driver of the whole Compose* toolset, see chapter 2 for more details. After all the preprocessing is finished by the other tools, SECRET is called via its run method. It receives the CommonResources as a parameter, this object contains the properties needed for SECRET. These properties are: the path of the XML specification file, if debugging should be turned on, where to place the output file and what system actors should be added to the resources after the filters. But most importantly it contains the list of filter modules to be parsed.

6.2.3 Conflict reporters

To be able to generate a decent error report the tool incorporates a conflict reporting class. This class uses a Strategy pattern to dispatch the reporting call to the correct output. This can be reporting conflicts on the commandline, generating an HTML file or some other format. For this thesis version I implemented a command line and an HTML version. If a new reporter is written it can be invoked by a parameter to the tool.
The first thing *SECRET* does is to call the XML parsers to start parsing the XML file and to fill the *SecretRepository* with the parsed information. When the parser is ready, the *AbstractVM* is initialized, all used maps are cleared. *SECRET* then starts iterating over every filter. For each filter it gets the actions of this kind of filter. Then for each of these actions the affected resources are retrieved. With this information the *ActionDescription* and *ResourceDescription* are created and added to the VM. If all of the filters are parsed and added to the VM the external entities are added to the same VM. Subsequently for every affected resources an *analyse* call is made to the VM which may or may not yield a conflict. If a conflict was detected this conflict is reported to the *HTMLReporter*.

### 6.4 Keeping Snow White healthy problem detection output

This section shows the output of the tool for the “Keeping Snow White healthy” problem. As stated in chapter 3, using meta filters create a lot of conflicts as we are not able to be specific. The output of the tool is an *HTML* file. A small part of this file given in figure 6.5.

### 6.5 Diner in time problem detection output

The output for the “Diner in time” problem is shown is figure 6.6.
6.5 Diner in time problem detection output

Figure 6.5: Keeping Snow White healthy problem detection output

<table>
<thead>
<tr>
<th>Violation on: timing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td>: System</td>
</tr>
<tr>
<td>: Scheduler</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Violation on: parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td>DietCheck.check : Meta</td>
</tr>
<tr>
<td>DietCheck_sync : Meta</td>
</tr>
<tr>
<td>: System</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Violation on: selector</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td>DietCheck.check : Meta</td>
</tr>
<tr>
<td>DietCheck sync : Meta</td>
</tr>
<tr>
<td>: System</td>
</tr>
</tbody>
</table>

Figure 6.6: Diner in time problem detection output

<table>
<thead>
<tr>
<th>Violation on: timing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td>PriceWantToEat : Wait</td>
</tr>
<tr>
<td>: System</td>
</tr>
<tr>
<td>: Scheduler</td>
</tr>
</tbody>
</table>
Conclusion and the road ahead

“The blood spattered bride...”

7.1 Related work

There are only a few papers that cover the conflicts between aspects. This is mostly due to the fact that without making changes to the source itself it is very difficult to reason about the source code of the aspects. One semantic aspect can be implemented in many different ways. Each extension of the language is however undesirable as semantic conflict detection tool is not a functional extension of the program. Semantic reasoning only detects conflicts present in the program, it does not add extra functionality. We do not want to clutter the aspects to accommodate the semantic reasoning.

The analysis of aspects oriented software is becoming increasingly popular. The use of aspects as self contained components can increase the ability to analyse the interactions between aspects. However much work has to be done in this area. The recently held workshop on the analysis
of aspect oriented software[19], discussed presented some types of conflicts that can occur with aspects. The proposed conflict types are largely covered by the semantic reasoning model presented in this thesis. This includes; conflicts between aspects, conflicts between the aspects and the base system.

R. Douence, P Fradet and Südhold[31] analyse possible conflicts in stateful aspects. They have defined their own formal framework where aspects are defined. They have made the variables of aspects explicit in this framework. Using these shared variables they identify the interactions between aspects. They can automatically resolve these conflicts by reordering the aspects, they use newly defined operators for this. Interactions arise when separate aspects are composed on the same joinpoint. By making the interactions at such joinpoints explicit, they are able to reason about state conflicts. Chapter 3 showed that the conflicts often can not be encapsulated into one variable. The authors only look at the direct interactions between the aspects and not the side effects the aspects may have. The in this thesis presented approach these side effects are modelled, the variables can be seen as a subset of the resources defined in this model. They only have a formal framework, but it has yet to be implemented in an AOP approach. SECRET does work with the Composition Filters approach on the .NET environment.

7.2 Discussion

The scalability of the tool is currently untested, I am however confident that SECRET will be able to scale as the order of SECRET is linear with the number of filters. If we would assume a mid-size application with 100 concerns. Presumably only 25 of these concerns are superimposed on the same join. At each of these shared jointpoints there will be on average 10 filters. However these 10 filters can, if not constrained, generate 10! distinct orderings. These orderings are only generated for the purpose of analysis, we could also use the ordering that is chosen for the run-time. Then only the orderings have to be analysed that use run-time constraints. As the filtering of the output, done by FIRE, is not yet implemented. The order of the filter mechanism is thus unknown. We do however know that is related to the number filters and will be at most squared.

The semantic detection model presented in this thesis, is generic and completely user adaptable. The resources are currently limited to read and write operations. These resources can be made more intelligent. Imagine a resource which has some maximum and a counter. Each operation can then increase the value of the counter until the maximum has been reached. If the counter exceeds the maximum a conflict is detected. The detection algorithm is then no longer based on regular expressions but on the values of operations. This extension would make the tool even more flexible and tailorable. The current abstract virtual machine should be extended to facilitate this.

The presented solution for detection the semantic conflicts between aspects can also work with other AOP approaches. If we specify the condition, matching, substitution and actions of a piece of advice we can do the exact same analysis given our resource model. This gives us an indication that the right abstraction has been made. Therefore the tool is reusable in different AOP approaches.
7.3 The road ahead

This section presents several future enhancements of the Composition filters and of the SECRET tool.

7.3.1 Detailed look at current filters

This thesis showed an analysis of the current filter types but it was not an exhaustive analysis and could be further enhanced. Certain properties may be of more interest than other properties. One could imagine that in a realtime system an system programmer would be more interested in timing and deadline properties than state properties.

7.3.2 Focus in the conflict reports

The tool currently detects all possible conflicts. This can be a quite large set of signaled conflicts. In a future extension of the tool one could introduce different levels of conflict severity. For instance warnings, errors and fatal errors. In the ConflictReporter such a filtering can be built in.

7.3.3 Increase reasoning power

Section 3.2.2 introduced a problem where two conditions were excluding one another. The tool was able to detect these problems. If the tool was able to reason about the conditions themselves, it could be even more specific than giving a conflict on the state. The use of the Object Constraint Language (OCL) can express certain behaviour. This approach has the potential of being declarative, so one can reason about it. To illustrate this fact I will now show both concerns again but now with the OCL syntax. The entire implementation part is removed.

In OCL one can express certain constraints like invariants, pre- and postconditions. The Diet concerns specified that the amount to eat may not be bigger than 6. One possible OCL expression for this constraint is shown in listing 7.1.

```
context Snowwhite :: eat ( amount : int , type : int )
inv : ( self.food() + amount.of.food ) <= 6;
```

Listing 7.1: Diet precondition in OCL

In OCL it is only possible to read the state of variables and system attributes. These expressions are therefore executed without side-effects. Therefore we can use conditions to check whether certain properties hold. The Diet aspect is rewritten to use conditions and an error filter, to express the same behaviour as the original aspect. This code is listed in listing 7.2.

```
concern Diet
begin
  filtermodule dietcheck begin
    conditions
      not.too.much = ((self.food() + amount.of.food) <= 6));
    inputfilters
      check : Error = { not.too.much => eat };
```

Pascal Dürr
Conclusion and the road ahead

The above listing clearly shows the reduction of code and the more effective use of the composition filter constructs. By itself the in OCL expressed conditions do not increase the reasonability for SECRET. However if we have means to express the OCL expression in terms of resources we could be able to identify the conflict on a variable in the OCL expression. A tool could analyse the OCL syntax and generate the resources and the operations on these resources. It subsequently would create a new instance of the filter and assign the appropriate filter actions to it. Then SECRET would be able to give a precise specification of the conflict.

To make the example complete the EnoughProtection aspect is also given with OCL expressions in listing 7.3.

There are two remarks that need to be made when using OCL in the filter specifications. The first is the loss of context information. If you compare the full OCL code snippet to the one used in the example, you will find that the context information is missing. This context information is dynamically assigned in the filter environment, but it should be checked if certain self calls really exist on all superimposed objects. Secondly the runtime part of the Compose* toolset should be extended to facilitate an OCL interpreter. This interpreter is currently not implemented.

7.3.4 Traversing down a dispatch chain

Imagine a filter set, $A$, where one filter dispatches to another object. This dispatch call may result in evaluation of another filterset, $B$. The combination of the filterset $A$ and the dispatched filterset $B$ may also introduce conflicts. To be able to traverse a dispatch chain a clear picture
of all the possible paths and dependencies should be present. One could imagine that the superimposition analysis, combined with the signature generator, could provide this graph almost automatically. The dependencies are also present when using meta filters which can call other objects. The result can be a graph, where the nodes are objects or concerns and the edges are labelled with relation name, e.g. dispatch or meta call.

With this graph the entire composition of the dispatch flow can be calculated. For each possible path a tool like SECRET can do its analysis.

### 7.3.5 Development of new filters

New filters can influence the behaviour of the superimposed system. For instance by adding atomic transaction handling and panic mode behaviour.

The atomic transaction filter can be used to specify which messages should be executed directly after each other. This combination of messages can be seen as one atomic action. If a certain message fails, the entire combination is rolled back.

In panic mode the behaviour of the base system depends on the total system load at a certain moment. If the load increases some messages will be skipped or postponed until later.

### 7.3.6 Analysis of dynamic behaviour

The tool currently operates in a static context. One could imagine that the same model implementation runs in parallel with the runtime execution of the underlying system. Then the resource usage can be dynamically determined and used to avoid conflicts before they occur. These conflict avoidance algorithms, similar to deadlock avoidance algorithms could be introduced to create a safer execution environment. In the runtime we can add monitors to the resources. Each time a filter is executed the operations linked with that filter are carried out on the resources. The monitor detects this update and check whether adding these operations result in a conflict. If not the call is allowed to pass, otherwise the message is rejected.

### 7.3.7 Introduce concurrency

SECRET currently only deals with single threaded environments. If the runtime is in a multi threaded environment, new types of conflicts may arise, like deadlock, starvation and live lock. These conflicts can be detected by giving each thread its own AVM and let these AVMs share the common resources used by more than one thread.

In distributed environments the same approach is valid. One could introduce one extra AVM to hold the global resources.
### 7.4 Conclusion

In this thesis we have developed a generic model, in order to detect semantic conflicts between aspects, expressed in Composition Filters. This model allows us to reason about the conflicts at a higher level of abstraction. At this level we can identify the areas where the conflicts occur as these are not captured easily by one filter on its own.

After analysis of the current filter types and the conflicts that occur between these filters, filteractions, resources and the operations on these resources have been identified. This analysis holds for the current set of filters types. We envision future specializations of these elements as more analysis of filters is carried out. This model, its instantiation and implementation as a tool is sufficiently generic to incorporate such future extensions.

Our tests on the available examples show that the tool was able to identify all conflicts. The realisation of the tool functions as a proof of concept and validates the model.
Bibliography


Conclusion and the road ahead
Example XML filter specification

This XML file is an example of a filter specification file which is the input for SERCET.

```xml
<?xml version='1.0' encoding='utf-8'?>
<secretfilterdefinition>
  <filters>
    <filter type="Error">
      <accept> <action type="continue"/> </accept>
      <reject> <action type="exception"/> </reject>
    </filter>
    <filter type="Dispatch">
      <accept> <action type="dispatch"/> </accept>
      <reject> <action type="continue"/> </reject>
    </filter>
    <filter type="Wait">
      <accept> <action type="wait"/> </accept>
      <reject> <action type="continue"/> </reject>
    </filter>
    <filter type="Substitute">
      <accept> <action type="substitute"/> </accept>
      <reject> <action type="continue"/> </reject>
    </filter>
    <filter type="Meta">
      <accept> <action type="meta"/> </accept>
      <reject> <action type="continue"/> </reject>
    </filter>
    <filter type="Realtime">
      <accept> <action type="setrtprop"/> </accept>
      <reject> <action type="continue"/> </reject>
    </filter>
  </filters>
  <actions>
    <action type="continue">
      <operation type="r"/>
    </action>
  </actions>
</secretfilterdefinition>
```
Example XML filter specification

```xml
<operation>
  <operation_type="w"/>
</operation>

<action type="exception">
  <operation_type="r"/>
</operation>

<operation>
  <operation_type="w"/>
</operation>

<action type="dispatch">
  <operation_type="r"/>
</operation>

<operation>
  <operation_type="w"/>
</operation>

<action type="substitute">
  <operation_type="r"/>
</operation>

<operation>
  <operation_type="w"/>
</operation>

<action type="wait">
  <operation_type="r"/>
</operation>

<resource name="timing"/>
</operation>

<action type="setrtprop">
  <operation_type="r"/>
</operation>

<resource name="deadline"/>
</operation>

<action type="meta">
  <operation_type="u"/>
  <resource name="selector"/>
  <resource name="target"/>
  <resource name="state"/>
  <resource name="server"/>
  <resource name="sender"/>
  <resource name="parameters"/>
</operation>

<action type="System">
  <operation_type="r"/>
  <resource name="selector"/>
  <resource name="target"/>
  <resource name="state"/>
  <resource name="server"/>
  <resource name="sender"/>
  <resource name="parameters"/>
</operation>

<operation type="w">
  <resource name="timing"/>
</operation>

<action type="Scheduler">
  <operation_type="r"/>
  <resource name="selector"/>
  <resource name="target"/>
  <resource name="deadline"/>
  <resource name="timing"/>
</operation>
```
Listing A.1: Example filter specification