Designating join points in Compose* - a predicate-based superimposition selector language for Compose*

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

Wilke Havinga

May 23, 2005

Graduation committee:

prof. dr. ir. M. Aksit
dr. ir. L.M.J. Bergmans (supervisor)
I. Nagy, Msc.

Abstract

Aspect Oriented Programming is an increasingly popular approach used to increase the modularity of applications.

The Compose* project implements an aspect oriented programming language based on the .NET platform. By using the Common Language Infrastructure defined by .NET it is independent of a particular implementation language.

Using Compose*, it is possible to define concerns that superimpose filtermodules on existing base classes. Filtermodules can intercept and modify message calls, thus enabling concerns to change the existing behaviour of an application.

To define where filtermodules should be superimposed, Compose* uses a superimposition selector language. Problems with the current selector language are its limited expressiveness and the tight coupling between selector expressions and base classes, thus impeding the maintainability and evolvability of applications. This thesis describes a new superimposition selector language based on predicate logic. Selectors written in this language specify where filtermodules have to be superimposed based on the static structure of an application. The selector language is independent of any implementation language because it uses an abstract language model to represent the language model used in Compose*.

Additionally, the selector language implements support for metadata annotations, a feature found in recent programming languages such as Java and C#. Annotations can be attached in the source of base classes and used as a selection criterion in selector expressions, allowing for selection based on design information rather than application structure or syntax. This also removes the tight coupling between selector expressions and base classes and addresses the scattering of annotations over reusable base classes.

The combination of selection based on annotations and the superimposition of annotations allows for the derivation of annotations, a mechanism that removes the need to manually enumerate where annotations should be superimposed. However, it also introduces the possibility of dependencies between selectors. This can lead to conflicts, as different orders of evaluating the selectors and superimposing annotations can lead to different results. In addition, circular dependencies could potentially lead to non-termination of the selector evaluation mechanism. An algorithm is presented to handle the superimposition of annotations and evaluation of selectors in a correct order. The algorithm also detects dependency conflicts.

Contents

How to read this thesis

This page contains a short guide to reading this thesis.

Chapter 1 contains an introduction to aspect oriented programming and describes the evolution of programming languages. If you are already familiar with the concept of aspect oriented programming, it is safe to skip this chapter.

Chapter 2 is an introduction to .NET and our aspect oriented programming framework, Compose*. It explains the architecture of Compose*, and the tools we use to implement it. If you are already familiar with the internals of Compose*, it is not necessary to read this chapter.

Chapter 3 identifies the subject of this thesis: creating a superimposition selector language for the Compose* project. Chapter 4 describes the design of this selector language, while chapter 5 discusses its realisation. Chapter 6 explains how metadata annotations can be used to select elements of a program based on design information. Chapter 7 concludes the work.

A basic understanding of programming in predicate-based languages (esp. prolog) is strongly recommended for reading this thesis. In case of doubt, consult any introductory guide on prolog programming, for example [Brn99] (freely available on the internet).

Chapter 1

Introduction and Background

The first two chapters have originally been written by six MSc. students at the University of Twente. They have been updated for use in this thesis, reflecting the current state of development. These chapters serve as a general introduction into Compose* and the underlying techniques. The chapters are used in the master theses of these authors:

Compilation and Type-Safety in the Compose* .NET environment.
Frederik J. B. Holljen

Detecting semantic conflicts between aspects.
Pascal E. A. Dürr

Automated Reasoning about Composition Filters.
Raymond Bosman

Superimposition in the Composition Filters model.
Christian A. Vinkes

Towards Safe Advice: Semantic Analysis of Abstract Communication Types in Compose*.
Tom Staijen

Performing transformations on .NET Intermediate Language code.
Sverre R. Boschman

Designating join points in Compose*.
Wilke Havinga

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 paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [Wat90]. Functional languages try to solve significant problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

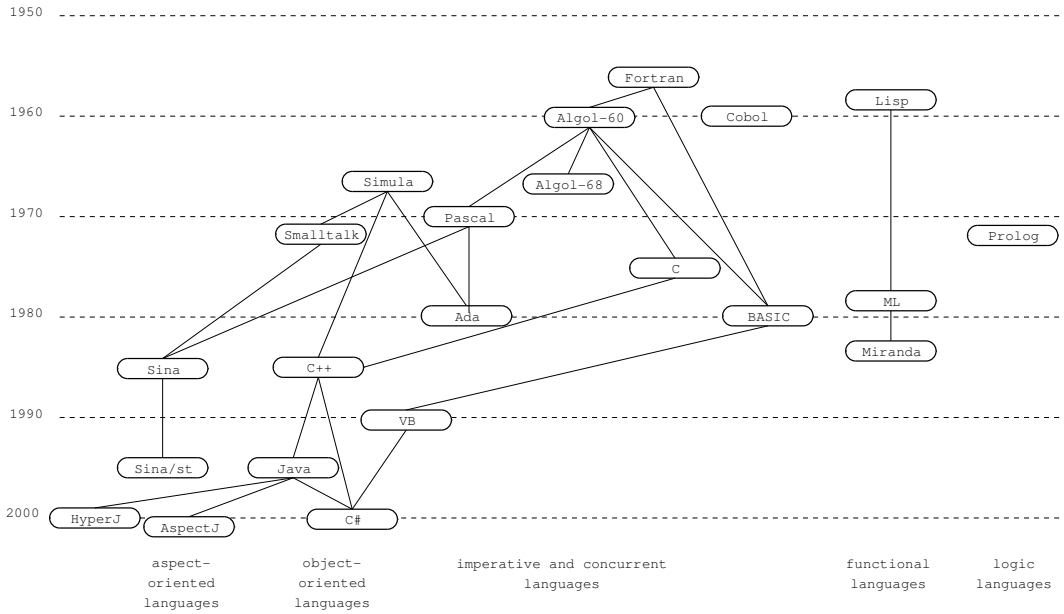

Figure 1.1: Dates and ancestry of several important languages

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [Wat90].

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 for it to become popular. The most well known object-oriented languages are C++, Java and Smalltalk [Wat90].

Aspect-Oriented Programming (AOP) is a paradigm that solves the problem of crosscutting concerns. 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 [KLM+97, Kim02]. 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.[GHJV95]:

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 concerns, 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 [Gre01]. As [OT01] et al. point out, in most cases formalisms such as programming languages and design notations only provide one prevalent means of decomposing software (they only support one dimension of concern). This causes problems when formalisms for the same software use different dimensions of concern. For example, requirements are often specified by a function or feature while object-oriented design and code is decomposed using classes. This creates a conceptual mismatch and requires developers to switch constantly between different representations of the same concept.

1.1.3 Example

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

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

Listing 1.1: Modeling logging without aspects

By adding the logging code to the class of object A, two concerns become mixed up in a single class (class A not only implements its "own" concern, i.e. adding two integer values, but also handles the requirements of a second concern, i.e. logging the performed operation). Crosscutting is the situation where a system requirement is met by placing code into different objects throughout the system [GL03]. 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 (same) 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 the problems described here will 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 [EFB01]. Such techniques are known as Post-Object Programming (POP) mechanisms [EFB01]. 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 would not have been 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 [AKLO01], i.e. objects can still be used. The fundamental goals of AOP are twofold [GL03]: first of all, to provide a mechanism for the description of concerns that crosscut other components. Secondly, to use this description to allow for the separation of concerns.

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

Listing 1.2: Modeling logging using aspects

Listing 1.2 creates a new aspect which executes the logging code after each call to addTwoIntegers. Line 3 specifies the pointcut, i.e. where to execute the aspect code; in this case, when the addTwoIntegers method is called on an object of class A. Line 5 specifies when to execute the code; in this case, after the completion of the addTwoIntegers method. Using aspects has several advantages over the previous code [EFB01][AKLO01]:

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 that uses AOP techniques can be thought of as 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 does not make a clear distinction between a component- and aspect part. This is called the symmetric approach. According to [AKLO01], 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. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter the compiler generates an intermediate or machine language output (depending on the compiler-type).

The advantages of using source code weaving are:

- High-level source modification. Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler.
- Aspect and original source optimization. First the aspects are woven 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, when using source code weaving it is not possible to add multiple inheritance to a single inheritance language.

Intermediate Language weaving overcomes these drawbacks.

Intermediate Language weaving

Weaving aspects through an intermediate language (IL) gives more control over the executable program than source code weaving, because it is possible to create combinations of intermediate language constructs that can not be expressed at the source code level. 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 using the source code (which may not be available).
- Faster compilation. Only modified objects should be recompiled by the native compiler and finally woven.
- Better separation of concerns in the compilation model. First all compiling is done, thereafter the weaving. The native compiler has no knowledge about the aspects that will be added.
- Adding aspects at load- or runtime. A special classloader or a runtime environment can decide and do 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 Microsofts .NET IL.
- More error-prone. Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e.g. inlining of methods).

Adapting the Virtual Machine

Adapting the Virtual Machine (VM) removes the need to weave the aspects. This technique has the same advantages as Intermediate Language weaving and can also overcome some of its disadvantages. Aspects can be added without re-compilation, re-deployment, and re-start of the application [PGA02, PGA03].

Unfortunately, modifying the VM has one major disadvantage. Using an 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 [EFB01] 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 provides its own 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 [Gre01] and Hyperspaces [OT01], which together with Composition Filters [BA01] are todays main AOP techniques. A detailed description of Composition Filters will be given in section 1.2.

The AspectJ approach

AspectJ [Gre01] 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. AspectJ tries to be compatible in four ways:

upward compatibility: all legal Java programs must be legal AspectJ programs.

platform compatibility: all legal AspectJ programs must run on standard Java virtual machines

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

programmer compatibility: programming with AspectJ must feel like a natural extension of programming with Java

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called *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:

```
aspect DynamicCrosscuttingExample {

    Log log = new Log();

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

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


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

Listing 1.3: A example of dynamic crosscutting in AspectJ

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

The code that should execute at a given joinpoint is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice that specifies where the additional code is to be 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 definitions 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:

```
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* [OT01] 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 separa-

tion 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* [TO00] 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 contain the code units that populate the hyperspace. One way to do this is by creating a hyperspace specification:

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

Listing 1.5: Creation of a hyperspace

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

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

Listing 1.6: A specification of concern mappings

The first line indicates that, by default, all units contained within the package are in the Kernel concern of the Feature dimension. The other mappings specify that any method named "trace" or "debug" address the Logging and Debugging concern respectively. One should note that later mappings override the first one.

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

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

Listing 1.7: Defining a hypermodule

As this example shows, a hypermodule consist of two parts. The first part specifies the set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies

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

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

1.2 Composition Filters

1.2.1 The Composition Filters approach

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

The Composition Filters (CF) model is an extension to the Object-oriented model. It is closely related to Aspect Oriented Programming, in the sense that with CF it is possible to model Aspects, but CF dates further back in time. The base concept in CF is that messages that enter and exit an object can be intercepted, and manipulated in various forms, modifying the way in which the object behaves. To do so, in the CF model, a layer called the *interface part* is introduced. The resulting model and its components are shown in Figure 1.2.
The most significant components in the CF model are the *input filters* and *output filters*. Each individual filter specifies a particular manipulation of messages. Various filter types are available for different types of manipulations. The filters together compose the behavior of the object, possibly in terms of other objects. These other objects can be either *internal objects* or *external objects*. Internal objects are encapsulated within the composition filter object whereas external objects remain outside the composition filters object, such as globals or shared objects. The behavior of the object is a composition of the behavior or its internal and external objects. In 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.

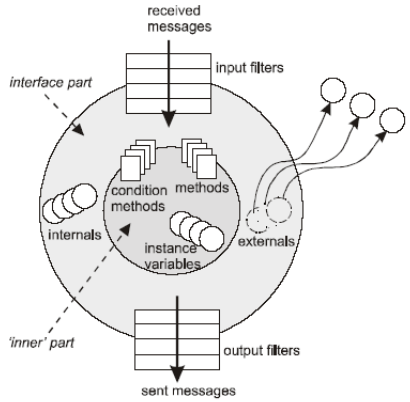

Figure 1.2: The components of the Composition Filters model

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

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

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

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

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

```
30      }
31  };
```

Listing 1.8: Example of a filtermodule specification

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

The superimposition mechanism

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

```
1   concern Tracing {
2
3       filtermodule tracingModule {
4           externals
5               log: Log;
6           inputfilters
7               logIn: Meta = ( isEnabled => [*.*] log.traceMessage );
8           outputfilters
9               logOut: Meta = ( isEnabled => [*.*] log.traceMessage );
10      };
11
12      superimposition {
13          selectors
14              withTracing = { C | isClassWithNameInList(C, ['Pacman', 'Ghost', 'World']) };
15          filtermodules
16              withTracing <- tracingModule;
17      };
18
19      implementation begin in "Java";
20
21      public class Log {
22
23          public void traceMessage(Message m) {
24              // tracing functionality here
25              // ..
26              // continue evaluating this message
27              m.fire();
28          }
29
30      }
31      end;
32
33  };
```

Listing 1.9: Example of a crosscutting concern

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

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 [Koo95].

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

1991 : The interface predicates 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 [Ber94].

1995 : The Sina language with Composition filters was implemented using Smalltalk [Koo95]. The implementation supported most of the filter types. Also this year, a preprocessor providing C++ with composition filters support was implemented [Gla95].

1999 : The Composition Filters language ComposeJ [Wic99] was developed and implemented. The implementation consisted of a preprocessor capable of translating Composition Filter specifications into the Java language.

2001 : ConcernJ [Car01] implemented as part of a M. Sc thesis. ConcernJ adds the notion of superimposition to composition filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

2003 : The start of the Compose* project.

Chapter 2

Compose*

2.1 Introduction

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

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

2.2 Overview of the .NET architecture

The .NET Framework is Microsoft's next step in the evolution of programming [Cor03a]. 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 [Cor03c]:

- 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 [Cor03c]: 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) [Wat00]. A detailed description of the CLR is given in section 2.2.1. The .NET Framework class library is a comprehensive collection of object-oriented, reusable types for .NET application developers. In section 2.2.2 a short description of the class library is given.

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

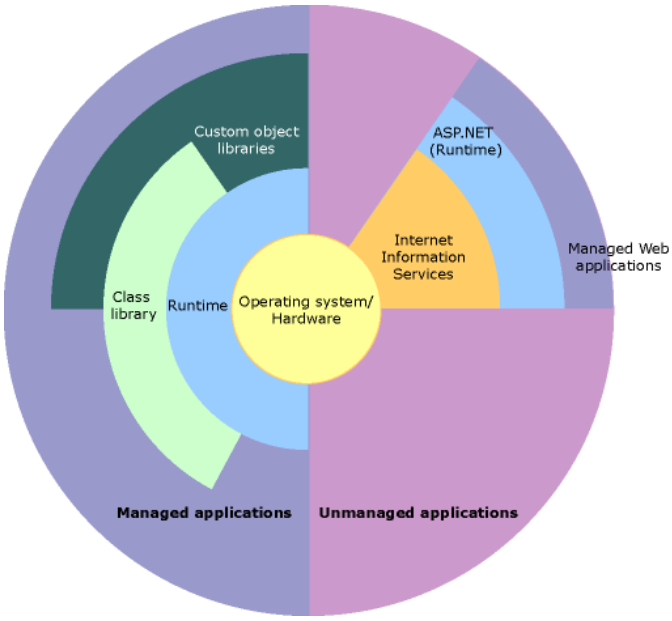

Figure 2.1: The context of the .NET Framework.
Source: Overview of the .NET Framework [Cor03c].

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 [Wat00]. 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 [Cor03b].

At the time of writing, the .NET framework is the only advanced Common Language Infrastructure(CLI) implementation available. A shared-source[1] implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [Stu02]. Also Ximian is working on an open source implementation of the CLI under the name Mono (`http://www.go-mono.com/`), targeting both Unix/Linux and Windows platforms. Another, somewhat different approach, is called Plataforma .NET (`http://people.ac.upc.es/enric/PFC/Plataforma.NET/p.net.html`) and aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

2.2.2 The .NET Framework class library

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

A wide range of common programming tasks (e.g. string management, data collection, database connectivity or file access) can be accomplished easily by using the class library. Also a great number of specialized development tasks are extensively supported, like:

- Console applications;
- Windows GUI applications (Windows Forms);
- ASP.NET applications;

[1] Only non-commercial purposes are allowed.

- XML Web services;
- Windows services.

2.2.3 Standardization

The entire CLI has been documented, standardized and approved [Int02] by the European association for standardizing information and communication systems, Ecma International.[2] Benefits of this standardization for developers and end-users are:

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

Interoperability is, for instance, achieved by using a standardized metadata and intermediate language (CIL) scheme as the storage and distribution format for applications. In other words, (almost) any programming language can be mapped to CIL, which in turn can be mapped to any native machine language.
The CLS is a subset of the CTS, and defines the basic set of language features that all .NET languages should adhere to. In this way, the CLS helps to enhance and ensure language interoperability by defining a set of features that are available in a wide variety of languages. The CLS was designed to include all the language constructs that are commonly needed by developers (e.g. naming conventions, common primitive types), but no more than most languages are able to support [Cor03d]. 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

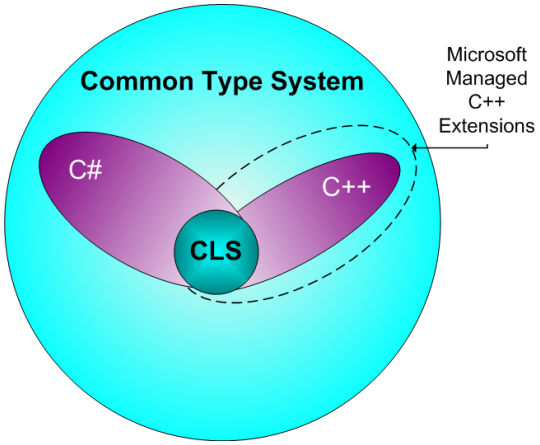

Figure 2.2: The relationships in the CTS.

[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.
[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.

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 class library is also available to the developers. Finally, the integration of the CLR with the platform it is running on is shown.

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.

- It is possible to specify how the superimposition of filtermodules can or should be ordered. This idea of being able to specify orderings on 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. The Consistency Reasoning Engine (CORE) is able to detect conflicts that may occur when a superimposition has been applied 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. In this case the latter filter is only reached with method m; this is consequently rejected and as a result the superimposition may never be executed. There are different scenario's that lead to these kinds of problems, e.g. conditions that exclude each other.
- The 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 indefinitely, 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 is similar to the 'around' advice in AspectJ, the current message is sent 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 behaviors pose a problem to the analysis tools. This feature specifies the behavior of the user object and offers an interface to the analysis tools to

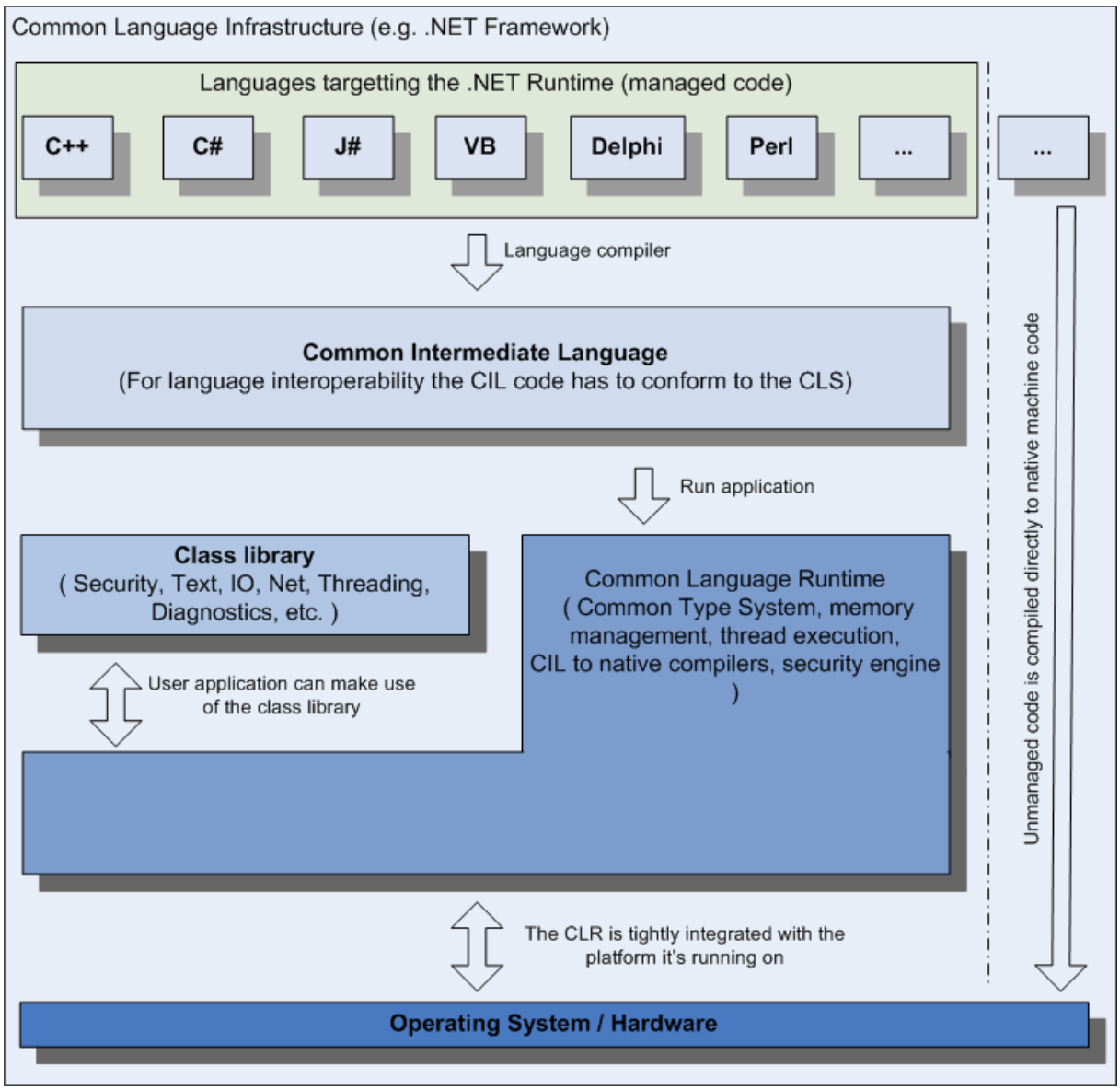

Figure 2.3: The main components of the CLI and their relationships.
The right hand side of the figure shows the difference between managed code and unmanaged code.

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* toolset, this section introduces a *Pacman* example. The *Pacman* game is a classic arcade game in which the user, represented by the *pacman*, moves in a maze to eat all the vitamins. Meanwhile the *Pacman* is being chased by *Ghosts*, these *Ghosts* will try to eat the *Pacman*. There are however four mega vitamins in the *Maze* that makes the *Pacman* über. In it's über state the *Pacman* can eat the *Ghosts*.
A simple list of requirements for the *Pacman* game is briefly discussed here:

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

2.4.1 The object-oriented design

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

Each class in figure 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 behaviour.

Pacman The *Pacman* class is the representation of the user controlled element in the game. It has some extra functionality, e.g. it stores whether the *Pacman* is über or not.

Ghost This is the representation of the ghosts chasing the *Pacman*. They have an extra property that indicates 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.

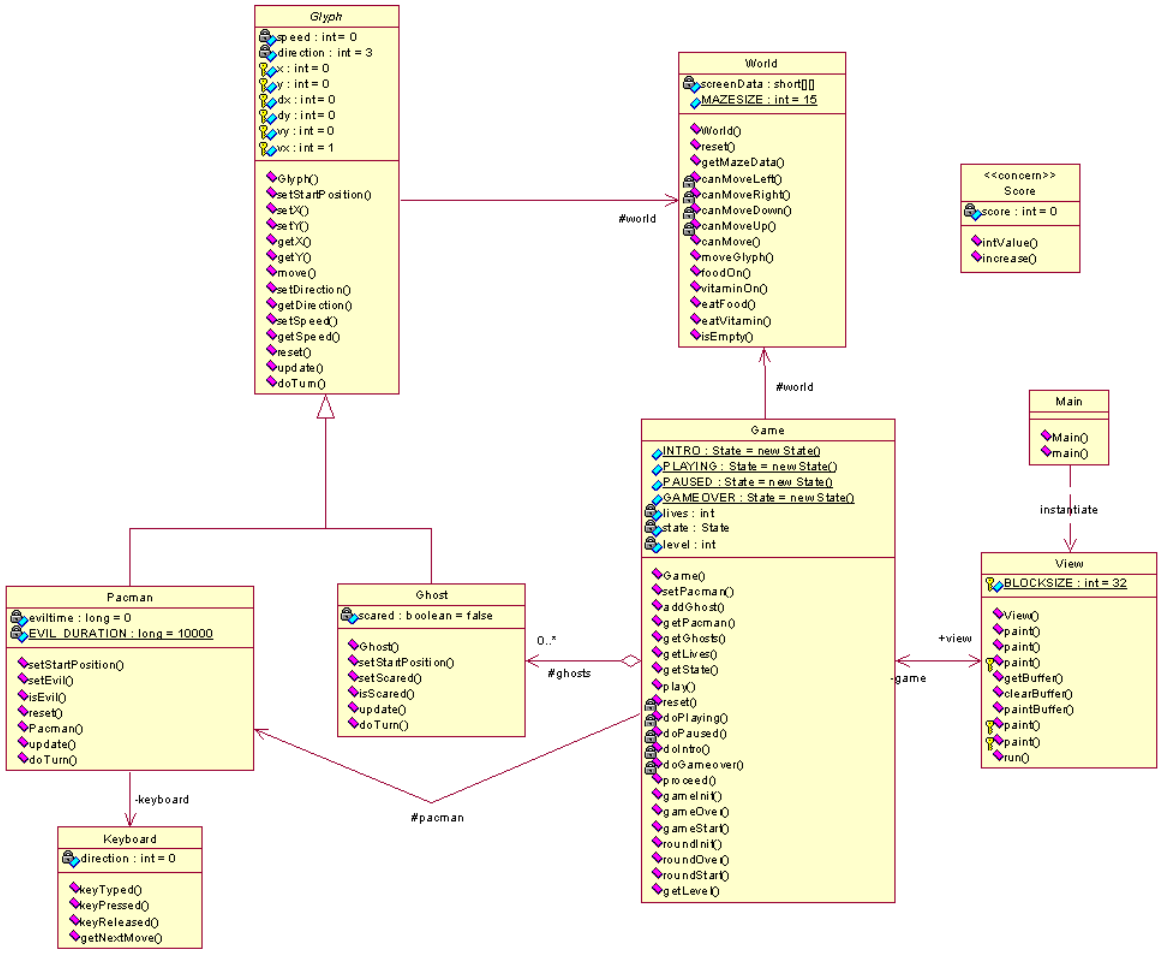

Figure 2.4: The UML class diagram of the object oriented Pacman game.

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 über. To satisfy this requirement, we create a *DynamicStrategy* concern that adapts the original *RandomStrategy*. This *DynamicStrategy* returns a move from class *StalkerStrategy* of *FleeStrategy*, depending on whether the *Pacman* is evil or not. The definition of this concern is given in listing 2.1.

```
1  concern DynamicStrategy in pacman
2  {
3    filtermodule dynamicstrategy
4    {
5      internals
6        stalk_strategy : pacman.Strategies.StalkerStrategy;
7        flee_strategy : pacman.Strategies.FleeStrategy;
8      conditions
9        pacmanIsEvil : pacman.Pacman.isEvil();
10     inputfilters
11       stalker_filter : Dispatch = {!pacmanIsEvil
12                                    => [*.getNextMove] stalk_strategy.getNextMove };
13       flee_filter : Dispatch = {[*.getNextMove]flee_strategy.getNextMove }
14   }
15
16   superimposition
17   {
18     selectors
19       strategy = { Random | isClassWithName(Random, 'pacman.Strategies.RandomStrategy') };
20     filtermodules
21       strategy <- dynamicstrategy;
22   }
23 }
```

Listing 2.1: DynamicStrategy concern in Compose*

The concern uses a *Dispatch* filter to intercept calls to the *RandomStrategy.getNextMove* method and redirect them to the method with the correct behavior. The *Dispatch* filter dispatches the call to either *FleeStrategy.getFleeMove()* or *StalkerStrategy.getHuntMove()*. If pacman is not evil, the call is forwarded to StalkerStrategy.getNextMove. Otherwise the call matches the next filter in the dynamicstrategy filtermodule and is forwarded to FleeStrategy.getNextMove. Calls to other methods within the RandomStrategy class are by default dispatched to the original *RandomStrategy* object, as we want to reuse the basic functionality of the *RandomStrategy*.

Another system requirement that needs to be added to the existing *Pacman* is the scoring mechanism. The score is updated each time the *Pacman* eats something, be it a vitamin, mega vitamin or Ghost. The amount of points awarded for eating an item 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 make it visible to the user. These events are all related to the score and are scattered over multiple classes: *Game*, *World*, *Main*. Therefore the score is identified as a crosscutting concern.

The *DynamicScoring* concern is divided into two parts. The first part is the concern definition, listed in listing 2.2. It intercepts each relevant call and sends the message in a reified form to the corresponding methods in the implementation.

```
concern DynamicScoring in pacman
{
  filtermodule dynamicscoring
  {
    externals
      score : pacman.Score = pacman.Score.instance();
    inputfilters
      score_filter : Meta = { [*.eatFood] score.eatFood,
                              [*.eatGhost] score.eatGhost,
                              [*.eatVitamin] score.eatVitamin,
                              [*.gameInit] score.initScore,
                              [*.setForeground] score.setupLabel }
  }

  superimposition
  {
    selectors
      scoring = { C | isClassWithNameInList(C,
                                  ['pacman.World', 'pacman.Game', 'pacman.Main']) };
    filtermodules
      scoring <- dynamicscoring;
   }
}
```

Listing 2.2: *DynamicScoring* concern

The *score_filter* filtermodule is superimposed on *Game*, *World*, *Main* in the *Pacman* example. The *scoreModule* is imposed on *Game*, *World* and *Pacman*. The *scoreView* modules is only imposed on *View*.

The second part of the *Score* concern is the implementation part listed in 2.3. This part receives the Reified Message and changes the score.

```
package pacman;

import Composestar.Runtime.FLIRT.message.*;
import java.awt.*;

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

  private Score() {}

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

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

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

```
30
31    public void eatVitamin(ReifiedMessage rm) {
32      score += 15;
33      label.setText("Score: "+score);
34    }
35
36    public void eatFood(ReifiedMessage rm) {
37      score += 5;
38      label.setText("Score: "+score);
39    }
40
41    public void setLabel(Label label) {
42      this.label = label;
43      label.setText("Score: ");
44    }
45
46    public void setupLabel(ReifiedMessage rm) {
47      rm.proceed();
48      label = new Label("Score: 0");
49      label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
50      Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo
51                                                    .getMessageInfo().getTarget();
52      main.add(label,BorderLayout.SOUTH);
53    }
54  }
```

Listing 2.3: *DynamicScoring* implementation

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

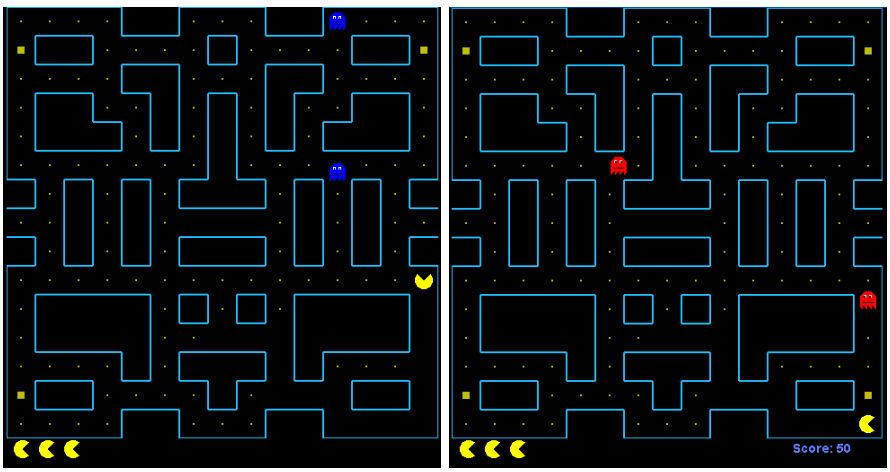

Figure 2.5: Pacman without and with concerns

2.5 Architecture

This section describes the architecture of the Compose* tool-set. We give a global description, followed by a more detailed architecture diagram and explanation.

The architecture of Compose* is divided into three main parts:

Visual Studio plug-in The Compose* compilation process is triggered by a Visual Studio .NET plug-in. Compose* projects are an extension of Visual Studio .NET projects.

Compose* Compile-Time The compile-time part of Compose* consists of modules that take care of the compilation of Compose* projects. The main module is called *Master*. It calls all other compile-time modules one by one. All the information gathered by these modules is stored into a central data store called *Repository*.

Compose* Run-Time The run-time part is responsible for the execution of compiled Compose* projects.

The three parts mentioned above are responsible for the compilation and execution of Compose* projects. The control flow between these parts is shown in figure 2.6.

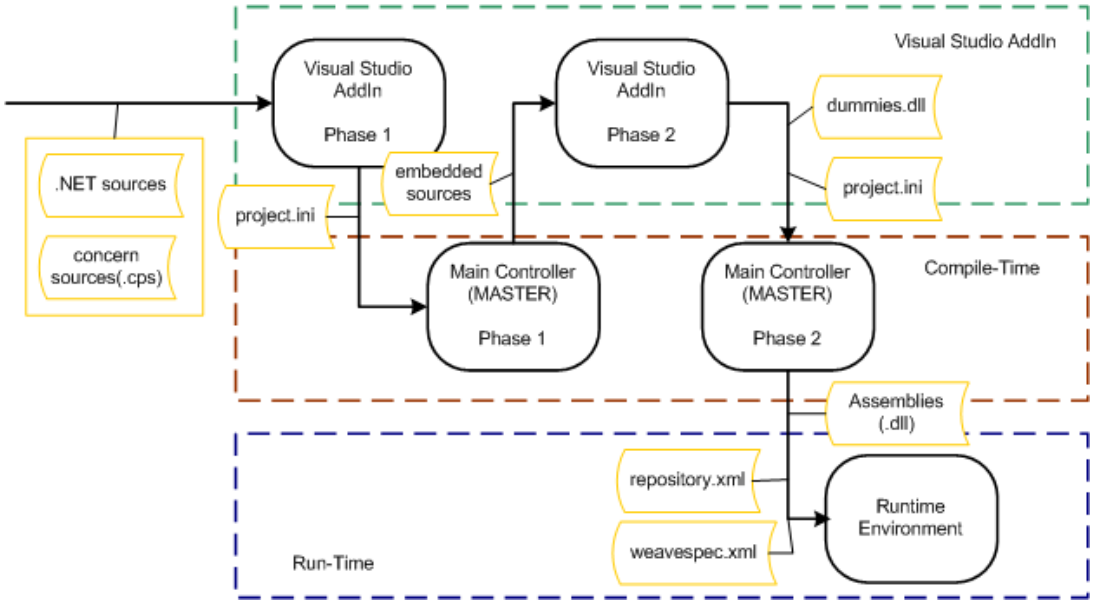

Figure 2.6: The Compose* control flow

The compilation process is initiated by a Visual Studio plug-in. In its first phase, the plug-in creates a project-specific configuration file describing the Compose* project. The project specific configuration file is then passed on to the *Master*, the heart of Compose* compile-time.

The *Master* obtains the location of all concern sources from the project-specific configuration file and extracts the embedded source code (parts of the concern that are written in the base language, e.g. J# or C#) from these concern sources.

After the embedded sources have been extracted, the plug-in continues with its second phase. In this phase the plug-in generates all information about the Compose* project needed by the compile-time. This information consists of:

- links to all project files, including locations of the (extracted) embedded sources
- which compiler to use to compile the project
- dummy sources of all .NET sources. The dummies reflect the static structure of the base program, but omits the implementation of methods. This is necessary to be able represent the static structure before the final version can be compiled.

This information is passed on to the *Master*. In the second phase of *Master* the sources are actually compiled, which results in:

- Compiled .NET sources
- Weave specification, containing execution joinpoints.
- *Repository* data serialized into XML.

The information gathered at compile-time is used by the Compose* run-time environment for successful execution of the compiled Compose* project.

In the following sections each part is described in more detail. Figure 2.7 shows a detailed architecture diagram, describing all the modules that Compose* consists of.

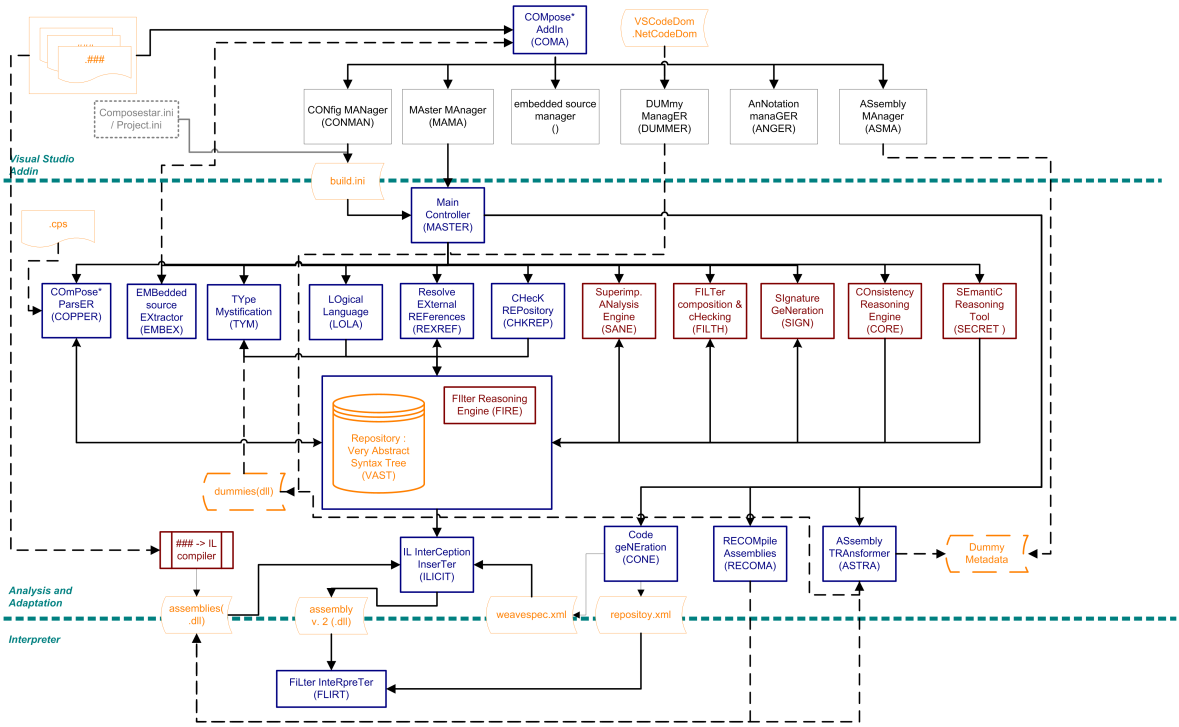

Figure 2.7: The Compose* architecture

2.5.1 Visual Studio Plug-in

As described in the previous section we distinguish between three levels within the Compose* architecture: the Microsoft Visual Studio plug-in, the compile time and the runtime level. The Visual Studio plug-in controls the compilation process. When the plugin is installed, it adds two menu items to the Visual Studio menu bar (see figure 2.8). This allows the developer to build or run his or her Compose* project from within the Visual Studio environment.

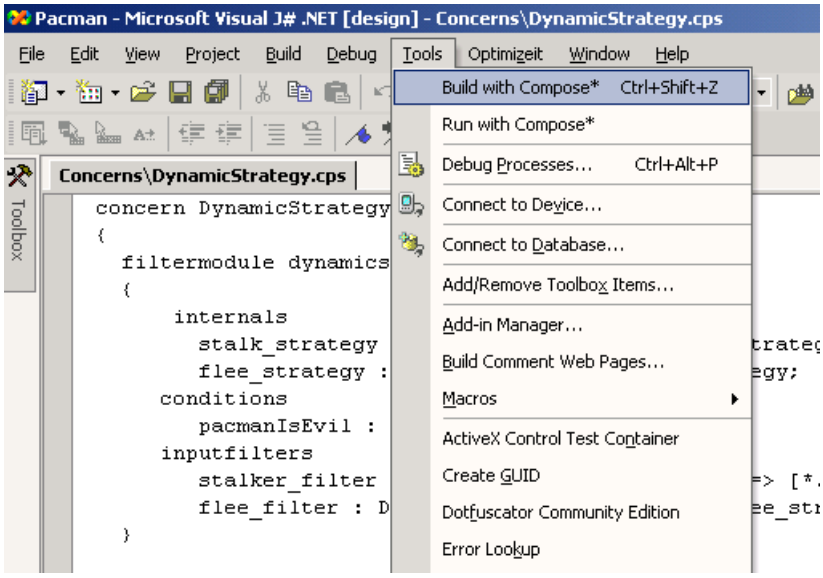

Figure 2.8: The Visual Studio plug-in

The plug-in performs several actions before it starts the actual compilation and runtime processes. It is divided into a number of components, visible in the upper region of figure 2.7.

First of all, the plugin is responsible for the configuration of all components involved in the actual compilation. The Config Manager *(CONMAN)* determines what files are part of the project, what language they are written in and which compiler should be used to compile them (e.g. the C# or J# compiler). It consults the *composestar.ini* file to obtain default settings for Compose*. The developper can override default settings by including a *project.ini* file with the project.

After the plug-in gathers all necessary information, it initiates phase one of the compilation by starting the Master Manager *(MAMA)*. This component calls the compile-time in a mode that parses the concern sources and extracts embedded source code from them. This first phase also gathers all the implementation details of classes in the project and creates dummy versions of each class that represents the static structure, but not the implementation. The sources of all implementation parts of concerns is extracted and put into separate class files. These files are added to the project and compiled. This task is handled by the Embedded Source Manager. The Dummy Manager *(DUMMER)* constructs the dummy files using the Code Generator of .NET in the language indicated by the configuration. These operations are required for a successful compilation of the complete source and aspect files.

If the first phase is successful and all dummy files are generated, the configuration is updated, so the compile-time will be able to find the dummies and compiled embedded source classes. Next, the Annotation Manager (ANGER) is called. This component extracts all the annotations from the original base classes and stores them in a XML file. This is necessary because the information about annotations is not available in the dummy versions of the classes.

The plug-in subsequently calls the Master Manager a second time to do the actual analysis and compilation of the Compose* project. Errors generated in either phase are shown to the developer using the Visual Studio information panes. If no problems occur, the Assembly Manager *(ASMA)* copies the generated binaries and other files needed by Compose* (e.g. the weave

specification and serialized repository) to the output folder.

2.5.2 Compose* Compile-Time

In this section, each module in the Compile-time is described by its inputs and outputs. It also gives a short description of each module.

Master

input: Configuration file (user provided)
description: Master is the initial module to be started when running the Compose* compiler. Master initializes the repository and loads the configuration file. It then proceeds by running the modules in the order presented.
output: If any of the modules detects an unrecoverable error they throw an exception. This exception is caught by Master and an error message is presented to the user.

COPPER (Compose Parser)

input: Concern sources (user provided)
input: Configuration file from Master.
description: The locations of concern sources for the project are extracted from the configuration file. Each concern source (.cps file) is parsed. Next, a tree walker extracts the elements from the parse tree and stores them in the repository.
output: A representation of the concern sources is stored in the repository.

EMBEX (Embedded Source Extractor)

input: Representation of concern sources (available through the repository)
description: The implementation part of concerns can be included in a .cps file, and can be written in any of the supported target languages. This module extracts the sources from the concerns, and creates new source files containing classes that can be compiled by the normal compiler for that implementation language.
output: Generated source files representing the embedded sources.

TYM (TYpe Manager)

input: Dummy sources (user provided)
input: Configuration file from Master
description: The locations of source files in 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.

REXREF (Resolve EXternal REFerences)

input: Repository data from COPPER and TYM pass one.

description: Concerns may have both internal and external references, e.g to methods or conditions. REXREF traverses the repository and makes sure that all such references are resolved.

output: Repository with internal and external references resolved.

LOLA (LOgic LAnguage)

input: Program elements and relations between them in the application, gathered from the repository (information supplied by TYM pass one).

description: Interprets superimposition selectors. Selectors are expressed in prolog and consist of (possibly complex) combinations of constraints on program element properties and relations.

output: Selected program elements (e.g. classes) for each superimposition selector.

CHKREP (CHecK REPository)

input: Repository with complete concern and meta-information data (from COPPER, TYM and LOLA).

description: The Repository Checker implements sanity checks on concerns. It checks for e.g. the existence of internal and external references to other concerns. This module is a work in progress.

output: Error messages in case problems are detected.

SANE (Superimposition ANalysis Engine)

input: Repository with complete concern and meta-information data (from COPPER, TYM and LOLA).

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

output: Repository with superimposition resolved.

FILTH (FILTer composition & cHecking)

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

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

output: Repository with filtermodule ordering resolved.

FIRE (FIlter Reasoning Engine)

input: Repository with filtermodule ordering resolved.

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

SIGN (SIgnature GeNeration)

input: Repository with filtermodule ordering resolved.

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

output: Repository with the full signature of all concerns.

CORE (COnsistency Reasoning Engine)

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.

ASTRA (Assembly Transformer)

input: Repository with complete information from all modules.
description: Updates the signatures of .NET dummy classes, taking (method) introductions by Compose* into account.
output: Updated meta-data describing the static structure of the application.

RECOMA (Recompile Assemblies)

input: Repository with complete information from all modules.
description: Compiles the original source code against the dummies. This is necessary as the original code may contain calls to methods that are introduced by Compose*, and thus only exist in the dummies (updated by ASTRA).
output: Updated version of the application assemblies, containing calls to dummy classes.

2.5.3 The Compose* Runtime environment

The Compose* Runtime environment consists of two layers: the JIT (Just In Time) layer and the Run Time layer. In the JIT layer ILICIT (IL InterCeption InserTer) inserts calls to 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 actually executing the concern code at execution joinpoints. Inside the Run Time layer we identify the Interception Handler and FLIRT (FiLter InteRpreTer). The Interception Handler is activated at execution joinpoints (as ILICIT inserted the necessary calls to the Interception Handler) and will dispatch the necessary calls to FLIRT to enforce the concerns.

A description of the modules involved:

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 FLIRT to enforce the concerns.
output: Modified IL code (existing in the .NET Runtime only).

FLIRT (FiLter InteRpreTer)

input: XML repository representation and dispatches from the IL code.
description: The responsibility of FLIRT is to provide runtime execution of Composition Filters. It creates an ObjectManager for each object as needed, and runs the appropriate filter code with it.

output: Execution of Composition Filters.

Chapter 3

Problem identification

This chapter describes the current state of superimposition selectors in Compose*, explains the issues we have with the current solution, describes requirements for an improved selector language, and elaborates on a solution proposal that satisfies these requirements.

3.1 Background

As described in chapter 1, aspect oriented programming (AOP) languages work by extending language mechanisms available in object oriented languages, or even by adding new language constructs. For example, a programmer can specify points in an application where the AOP framework will intervene (e.g. by intercepting messages between objects) to observe or change the original behaviour of the application[1].

Because the selection of certain points in the base application is central to the idea of AOP, it is clear that the language used for specifying such points (which are also called 'join points') is an important subject related to aspect oriented programming ("'A critical element in the design of any aspect-oriented language is the join point model."' [Asp04]).

Join points can be described by explicitly referring to places in the static program structure - for example: a method in a particular class. Methods and classes are usually referred to by their name or signature, pattern matching can be used to make this mechanism more powerful.

Compose* implements such a mechanism. The next paragraph gives an example and explains the shortcomings of the current selection mechanism with respect to important software engineering properties such as maintainability and evolvability.

3.2 Problem explanation

Consider the code fragment listed in 3.1. It specifies a 'Tracing' concern, which has the task to trace certain methods. Which methods are traced is specified in 2 stages: first, the filtermodule 'tracingModule' (line 2-7) specifies (at line 6) that it will match calls to all methods in the target object (notation: [*.*]), and redirect them (Meta filter) to the log.traceMessage method,

[1] This technique is commonly called 'static crosscutting'; AOP frameworks may also intervene based on runtime conditions or events (dynamic crosscutting) or by adding new elements to the application (introduction).

the implementation of which is not specified here. Secondly, the superimposition part (line 9-14) specifies on which objects this tracingModule should be superimposed, by listing the *exact names* of the classes on which the tracingModule will be imposed - in this example, we want to trace all calls to instances of the IncomingConnection and OutgoingConnection classes (line 11).

```
1  concern Tracing {
2      filtermodule tracingModule {
3          externals
4              log: Log;
5          inputfilters
6              log: Meta = ( [*.*] log.traceMessage );
7      };
8
9      superimposition {
10         selectors
11             withTracing = { *=IncomingConnection, *=OutgoingConnection };
12         filtermodules
13             withTracing <- tracingModule;
14     };
15
16     [..implementation..]
17 };
```

Listing 3.1: Tracing example

There are several problems with this approach:

- The superimposition selector refers directly to the names of certain classes, thus creating a tight coupling between the concern and the implementation classes. Tight coupling causes problems with relation to maintainability and evolvability (for a detailed discussion of these problems see [NBG+05]).
- It is hard to extend the program, because the programmer has to explicitly add every new class that should be traced to the list of selected classes. It is not possible to describe superimposition selectors in such a way that the concern can decide 'automatically' whether to trace classes that may be added in the future.
- In many cases we want to query based on a combination of several properties - for example, to select all the classes in the package 'java.util' that *also* implement the 'Serializable' interface. Such queries cannot be expressed in the current implementation.

3.3 Requirements

From the problems mentioned above we derive requirements for an improved selector language:

- It must be possible to describe a selector without referring directly to only specific classes, thus decoupling concerns from implementation classes. Better ways to describe join points can be based on namespaces, implemented interfaces, visibility, inheritance and other information that can be obtained from the static application structure.
- It must be possible to select program elements based on their intended *meaning* according to the application design. Information about the intention of parts of the program may be automatically derived (if possible) or supplied by the programmer. For example, if we can

write down the selector: 'all operations that may have an effect on the visual representation of the application on the screen', it would be possible to capture future extensions without explicitly adding them to a list in the concern.

- The selector language must be expressive, offering powerful language constructs that make it possible to write complex queries, e.g. by combining program element properties and relations in any conceivable way.

3.4 Solution proposal

By designing a language that can designate join points based on properties (such as the name or visibility) of program elements (such as methods, classes, packages etc.), as well as the relations between such program elements, we give the programmer a way to avoid direct references to only specific elements of the application, thus solving the problems caused by tight coupling between concerns and specific classes. In some cases, the static structure of the program can even be used to derive clues about the meaning of parts of the program (for example, all GUI classes might be gathered in the 'application.gui' namespace). However, using the namespace hierarchy, naming patterns (set/get-methods) or similar conventions that use the static structure of the program to convey information about the 'meaning' of parts of the application has its limitations and can even cause new problems, especially regarding the evolvability of the application [NBG+05], [TBG03].

Also, just creating a better selector language for querying static structure does not solve the tight coupling problem in all cases, as many queries will still refer directly to only specific program elements (classes).

Fortunately, recent developments in object oriented programming languages have opened up new possibilities to solve these problems. Custom attributes in .NET and Metadata annotations in Java 1.5 are techniques that allow a programmer to attach information about the semantics (meaning) of parts of the application. Although such metadata does not have any direct influence on the execution of the application itself, it can be used in selectors, thus removing the tight coupling between selector expressions and the application[NB04]. The use and possibilities of such metadata annotations will be explored in this thesis, and features relating to it will be included in the query language.

By using a suitable 'base' language that has the expressive power we need (i.e. it must be possible to express complex queries in this language, and easy to combine multiple properties/relations), we hope to overcome the limited expressiveness suffered by the current implementation[2]. Also, using an existing base language saves work and prevents us from reinventing the wheel.

Base languages that we considered are OCL, SQL or a standard predicate-based language using our own predicate library. These alternatives will be discussed in the analysis chapter.

3.5 Problem summary and conclusion

The current version of Compose* supports a simple superimposition selection mechanism that lacks features and causes problems with respect to several desired Software Engineering proper-

[2]There has been a proposal to use OCL as the selector language in Compose*. However, this functionality was never fully implemented - we will discuss the proposal to use OCL in chapter 4.

ties.

To address these issues we will present a generic and expressive predicate-based selector language.

- Generic means that the selector language is independent of any specific application implementation language. By basing our solution on a general-purpose language we hope to offer a powerful selector mechanism, that is also easy to extend in the future.
- Expressive means that the selector language offers powerful language constructs that enable us to write complex selectors, e.g. by combining multiple properties and relations.

The next chapter discusses our choice of a predicate-based language (as opposed to alternatives, such as OCL or SQL) and analyses how we can conveniently represent an application in such a way that we can query its static structure.

Chapter 4

Design of the selector language

This chapter explains the choice of using predicates as the basis of a selector language. It explains the design of a language for describing superimposition selectors that is independent of a specific implementation language. This is accomplished by defining a generic language meta-model, which is instantiated for a particular target language model (in our case, a representation of the .NET Common Type System). The elements found in a particular application are represented in a way that fits this meta-model. Finally, we propose a predicate-based language to query this model of the application.

4.1 Selecting a query language

There are several design alternatives for creating a language to query the application structure. In this section, we investigate these alternatives and discuss our choice of a particular solution.

The Compose* repository represents the static structure of the application. We view this structure as a 'database' that the user can write queries against. In that sense, the query language is quite generic. Therefore, we look at existing query languages and approaches that are relevant to the subject of querying static program structure.

4.1.1 Object Constraint Language (OCL)

Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) specification. It can be used to express constraints, assertions and pre/post-conditions on objects and relations between objects. To specify what objects a particular constraint should apply to, a set of elements has to be selected from the application model. Thus, an important part of OCL deals with querying the (UML) model of the application. The OCL specification mentions that OCL can be used 'as a query language' (section 7.1.1 of the OCL specification [OMG03]).

Because UML is often used to model (among other things) the static structure of applications, OCL seems like a logical choice as a language for selecting program elements. It has the expressiveness that we require.

Although OCL looks like a promising choice, we discovered that using it would lead to some problems in our case. These are our most important considerations:

- From earlier experiments we discovered that (at least for our uses) OCL expressions would

often become deeply nested; writing down and understanding these expressions was not very intuitive. Although OCL has the expressiveness that we need (we could not come up with queries that could not be expressed in OCL), it is not always possible to write them down concisely. One of the causes is that iteration has to be written down explicitly, which results in deeply nested OCL expressions. To demonstrate the problem, suppose that we want to select all classes in the 'gui' namespace that have a 'paint' method. Here is an OCL expression that accomplishes this:

```
1 AllTypes->select(c | (c.Namespace.toString().equals("gui")) and
2                      (c.Methods->collect(m | m.toString().equals("paint")).size() > 0));
```

- Recursion is often used to describe (nested) hierarchical relations such as the inheritance between classes or the architectural hierarchy of an application (namespaces/subnamespaces/classes). Unfortunately, it is not possible to define recursive queries (i.e. queries that refer to themselves) in OCL. Although OCL operations and 'let' expressions can be recursive[1], having to write custom operations for every case where recursion is involved is quite inconvenient when compared to languages where recursion is part of the standard language mechanisms.

- We would need to stretch or even break the intended meaning of the original OCL specification to implement certain features: although we could use the OCL syntax, we would have to (at least) create a customised set of operations to query the Compose*-specific model of an application. This is clearly not in line with the intention of the specification, which states that "[..] the full [OCL] specification can be used with the UML only" ([OMG03] , chapter 2, 'Conformance'). Implementing standards in (slightly) incompatible ways is considered harmful.

These considerations led us to believe that using OCL as the basis of our selector language would not be the best choice. As can be seen from the arguments above, this is an engineering decision; there are no strong conceptual motivations not to use OCL.

4.1.2 Structured Query Language (SQL)

The Structured Query Language (SQL) is *the* standard query-language used by almost every relational database system in use nowadays. If not for any other reason, it is already an option worth considering just because it is so well-known: many potential Compose* users will already be familiar with the language.

A model representing the static program structure can quite easily be mapped onto a relational database model. However, SQL is based on relational calculus, which is based on first order predicate calculus. The added value of using a SQL-engine rather than a generic predicate-based language is its specialisation on databases (query optimisation, indexing, ACID properties etc.) - "As the name implies, SQL is designed for a specific, limited purpose" [Wik05]. This is not the kind of specialisation we are looking for within the context of Compose*.

We could use only the SQL syntax, but that would remove most of the added value over using just a plain predicate-based language. Additionally, many queries can not be written down concisely in SQL. This is partly caused by the obligatory use of several keywords in each query

[1] even then the specification is not 100% clear about the semantics, as discussed in e.g. the Amsterdam Manifesto on OCL [CKM+99]

(select, from, where, ..), but also by the need to create explicit cross-reference tables (visible to the user) to represent many-to-many relationships between elements. The same example that was used in the OCL section above could be written down in SQL as follows:

```
1 SELECT c.id FROM Namespace ns, class c, method m
2 WHERE ns.name = 'gui' AND  c.parentNamespaceId = ns.id AND
3       m.parentClassId = c.id AND m.name = 'paint';
```

An important notion is that SQL does not allow for arbitrary reasoning, as it is not a turing complete language. This is a strong point as well as a weakness: reasoning about properties of a program written in a turing-complete language is hard or impossible. For example, it is impossible to determine whether such a program will ever terminate. This is is clearly not a desired property of a query language. However, turing-complete languages allow for more powerful reasoning.

To summarise, SQL can be seen as an extension of predicate logic, specialised for use within the domain of relational databases. We do not need this specialisation and the overhead it imposes on the syntax of queries. Therefore, we decided that using SQL would not be the best choice either.

4.1.3 Predicate-based queries (Prolog)

From the above, we can conclude that we are in fact looking for a very generic language that can be tailored to fit our own needs. The option that we chose is to use an existing logic predicate language: Prolog.

There are several reasons for using a predicate-based language:

- By using a generic language we avoid any domain-dependant 'a priori' restrictions.
- Using a turing-complete languages makes it possible to do arbitrary reasoning about the static application structure [Gyb02].
- Logic programming languages are declarative rather than imperative. This makes it easier to read predicate expressions [Gyb02], as they only have to specify the constraints on *what* to select instead of explicitly describing *how* elements should be selected (e.g. by explicitly navigating the structure like in OCL).
- Unification and recursion are very powerful concepts, convenient for expressing complex queries on the structure of an application. Unification is convenient for expressing relations between elements of the application, while recursion is convenient to walk through hierarchical structures, such as the inheritance relation between classes or the architectural hierarchy.
- Different ways to look at the static structure can be defined as (recursive) predicates that act as 'views' on the structure (similar to views in SQL). This makes it easier to write complex queries.
- The syntax is easy to learn (and also easy to parse).

To demonstrate the easy of using unification and the conciseness of this predicate notation, we write down the same selector as in the previous sections:

```
1 isNamespaceWithName(NS, 'gui'),
2 namespaceHasClass(NS, C),
3 classHasMethod(C, M),
4 isMethodWithName(M, 'paint').
```

Note that for the sake of clarity, we chose to use a rather elaborate naming scheme for the predicates.

We decide to base our implementation on prolog because it is the most commonly used predicate-based language, so potential users may already be familiar with its syntax. A lot of information about prolog is available on the internet, as well as several free/open source prolog interpreters.

4.2 Language models

To designate join points in an application, we need a clear way to describe which points we want to select. An important way to describe such points is by using the static structure of the program. Every application consists of a (usually complex) combination of program elements of the implementation language. In object oriented languages, such program elements are for example classes, methods, fields, parameters, and so on.

A logical first step in reasoning about properties of and relations between program elements is to define an abstract meta model that can represent the different kinds of program elements and their properties and relations.

Next, we derive a concrete meta model that captures the most important program element types of the .NET platform. The model is specific to .NET because it imposes (as compared to the abstract model) constraints on possible unit types and relations between them.

We finish this chapter by defining a convenient mapping of the meta model to a logic predicate language, which can then be used to query the model. Some examples are included to clarify how the mechanism works.

4.2.1 Different language, different model?

Language unit types vary quite a lot, depending on the language that is used. For example, C++ and C# use the concept of 'namespaces' while in Java they are called 'packages'. In this case, the concepts are identical, only the name is different. However, some unit types are specific to a certain language or framework, e.g. .NET has a unit type 'Delegate', while Java does not have a similar concept. Possible unit properties can be different as well: for example, most languages support the concept of unit visibility (public, private, protected) and other modifiers (e.g. synchronized, final, const) - but the actual names and meaning of such attributes can differ. Even the possible relations between program elements have different names or meaning. For example, C++ supports the concept of 'friend' classes, most other languages do not use such a concept.

Because we want our model to be language independent, we start by defining a simple, abstract meta model. This model offers a generic structure that can be used to represent all language-specific cases.

4.2.2 Abstract meta model

Fortunately, a common ground can be determined that captures the most basic properties of program elements. The goal in this section is to identify those properties, giving us a very abstract meta model that can capture almost any unit structure and properties that are in use by current programming languages.

Figure 4.1 shows a diagram of the abstract meta model. The elements of this model are explained below.

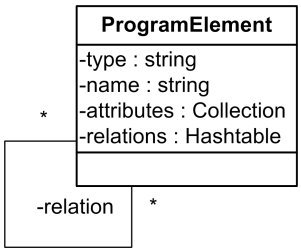

Figure 4.1: Abstract meta model

Program elements can generally be described by the following properties:

- Type, identifying the kind of program element, e.g. Class, Package, Method, Constructor, Field, Parameter, Localvar, ..
- Name, identifying a particular program element, e.g. java.util, Hashtable, getSomeValue, myVariable, ..
- A collection of fixed language attributes (represented by strings), such as visibility modifiers (public/protected/private) and other modifiers such as 'final', 'synchronized', 'static', and so on. To keep this model simple, we do not distinguish between several types of modifiers; they are just keywords that are either attached or absent. Obviously, some combinations can not occur in practice (e.g. public and private keywords attached to the same program element), but we leave such restrictions out of this abstract model.
- A hashtable, containing relations to other program elements. Each key into this table represents the name of a relation, while the corresponding value contains a collection of connected program elements. For example, a Class-type program element can have an 'implements' relation to a collection of program elements of the Interface type.

This model is deliberately kept very simple, as this allows us to design a small set of 'core' functionality that enables us to build a more complicated concrete meta-model (e.g. a meta-model representing .NET program elements) on top of it.

4.2.3 .NET specific meta model

Now that we have an abstract model, we try to establish a list of concrete unit types, possible properties and the relations between units for the .NET framework. This gives us a concrete language meta model that can be used in Compose*.

It should be noted that even this language model is largely language independent: as explained in the Compose* chapter, all language compilers adhering to the .NET Common Type System will generate code that can be represented by this .NET specific model.

Figure 4.2 shows a meta model of the most important units that can be present in a .NET program. Each rectangle in this diagram represents a particular type of program element. The diagram documents the .NET meta model that we use to implement our selector language, describing the possible relations between program elements.

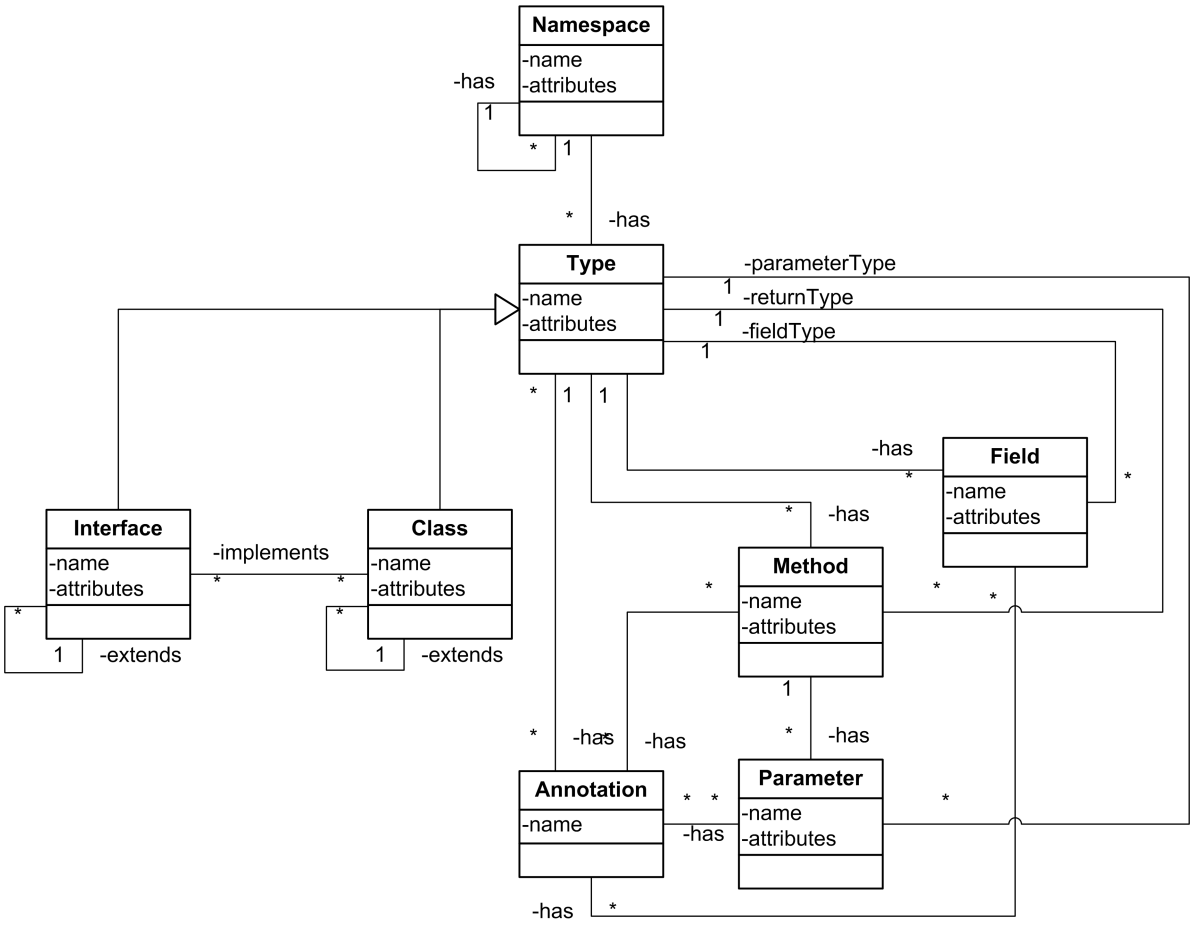

Figure 4.2: .NET meta model

As can be seen in the diagram, namespaces can contain child namespaces and types. To represent this in terms of the abstract model, there is a 'ChildType' relation from a program element with type 'Namespace' to another program element of type 'Type'. Each relation is defined in both directions, so there is also a relation 'ParentNamespace' from the Type to the Namespace. In other words, all relations are bi-directional, but not symmetrical.

The added value of this concrete model is that it puts constraints on the possible relations, the types of elements that can be on the other end of each relation, and the multiplicity of each relation. For example, a namespace can contain an arbitrary number of types, but any type always belongs to exactly one namespace.

There exists one special program element type in this diagram: the Type element is a supertype

of both interfaces and classes, which means that both 'Class' and 'Interface' elements are also 'Type' elements. In some cases we just want to select all types, while in other cases we may be interested in the 'implements' relation between class and interface. This works just like inheritance in object oriented languages.

The same distinction could have been made to separate methods into a generic type 'Operation', subtyped into 'Constructors' and 'Methods'. We chose not to do this because it adds complexity[2], while the representation of constructors is the same as for methods (in most cases using a particular naming scheme). Constructors are therefore represented as normal methods in this model.

The Annotation program element type represents metadata annotations in .NET, which can be attached to types (classes/interfaces), methods, fields and parameters. The use of annotations will be explained in chapter 6, at this point it suffices to know that they are represented like any other program element, and that the 'attachment' of an annotation to a program element is represented like any other relation between program elements.

4.3 Mapping to predicate logic

Now that we have defined a concrete .NET meta model and its properties, we want to express it in terms of logic predicates. The mapping between the predicates defined here and the diagrams in the previous sections is pretty straightforward, at least conceptually. How we make the connection between an object oriented repository of program elements and a logic programming language in practice is another matter, which will be discussed in the 'design and implementation' chapter (chapter 5).

The predicates are expressed in prolog. As a reminder, all prolog predicates start with a lowercase character, while variable names in prolog start with a capital letter. String constants are put between single quotes ('this is a constant').

4.3.1 .NET language units

These basic .NET language unit definitions reflect the structure in figure 4.2

```
1  isNamespace(Namespace).
2  isInterface(Interface).
3  isClass(Class).
4  isAnnotation(Annotation).
5  isField(Field).
6  isMethod(Method).
7  isParameter(Parameter).
8
9  isType(Type) :- isInterface(Type).
10 isType(Type) :- isClass(Type).
```

For every unit type defined in the .NET model there is one predicate that selects all the units of this type. Evaluating the prolog query "isClass(C)" will thus result in a list of possible values for the variable 'C', where every class in the application will be returned as a possible value.

[2] if we make this distinction, what about destructors and/or finalize methods? etc.

The 'isType' definition expresses the inheritance between Class/Interface and Type. Although there can be no concrete instances of Type (it is an abstract unit type), the definition of isType instead returns both interfaces and classes. The two rules defining this predicate act as 'or' expressions - a program element is a type if it is either a class or an interface. The values that are bound to the variables are program elements, as defined in the abstract model. These are represented in prolog as internal objects that cannot be accessed directly - they have to be accessed through other predicates. As described in the abstract model, the properties of program elements are: type, name and attributes. We encode the type in the name of each predicate, as this keeps the language more concise[3]. Therefore, we only have to define predicates to get to the 'name' and 'attributes' properties.

4.3.2 Properties of program elements

The following predicates are defined to reach the properties of specific program elements:

```
1  isInterfaceWithName(Interface, InterfaceName)
2  isInterfaceWithAttribute(Interface, AttributeName).
3
4  isClassWithName(Class, ClassName).
5  isClassWithAttribute(Class, ClassAttribute).
6
7  isFieldWithName(Field, FieldName).
8  isFieldWithAttribute(Field, FieldAttribute).
9
10 [..and so on for the other language unit types..]
11
12 isTypeWithName(Type, TypeName) :- isClassWithName(Type, TypeName).
13 isTypeWithName(Type, TypeName) :- isInterfaceWithName(Type, TypeName).
14
15 isTypeWithAttribute(Type, TypeAttribute) :- isClassWithAttribute(Type, TypeAttribute).
16 isTypeWithAttribute(Type, TypeAttribute) :- isInterfaceWithAttribute(Type, TypeAttribute).
```

For example, we can select only the 'java.util' namespace by executing the query "isNamespaceWithName(NS, 'java.util').". This will render exactly one result value ('binding') for the variable 'NS' (assuming that this namespace actually exists in the application), which is the internal representation of this particular namespace. As another example, we can select all public classes by executing the query "isClassWithAttribute(C, 'public').".

Our next task is to define the relations between program elements.

4.3.3 Relations between program elements

These are the definitions of relations that our .NET meta model supports - these represent the lines between program element types in figure 4.2:

Namespace relations

```
1  namespaceHasInterface(Namespace, Interface): the Namespace contains Interface.
2  namespaceHasClass(Namespace, Class): the Namespace contains Class.
```

[3] imagine writing the alternative: "isClass(C), elementHasName(C, 'Myclass')" repeatedly in one query..."

```
namespaceHasType(Namespace, Type): the Namespace contains Type.
isParentNamespace(ParentNS, ChildNS): ParentNS is the parent namespace of ChildNS.
```

The root namespace (which has the empty string as its name) does not have a parent namespace.

Type relations

```
typeHasAnnotation(Type, Annotation): the Type has Annotation attached.
typeHasAnnotationWithName(Type, Annotation): the Type has an annotation with the specified
                                             name attached.
isSuperType(SuperType, SubType): the SubType directly inherits from SuperType.
typeHasMethod(Type, Method): the Type declaration contains Method.
```

Class relations

```
classHasAnnotation(Class, Annotation): the Class has Annotation attached.
classHasAnnotationWithName(Class, AnnotName): the Class has an annotation with the specified
                                              name attached.
classHasMethod(Class, Method):      the Class has Method.
classHasField(Class, Field):        the Class has Field.
isSuperClass(SuperClass, SubClass): the SubClass directly inherits from SuperClass.
classImplementsInterface(Class, Interface): the Class implements Interface.
```

Interface relations

```
isSuperInterface(SuperInterface, SubInterface): the SubInterface directly inherits
                                                from SuperInterface.
interfaceHasAnnotation(Interface, Annotation): the Interface has Annotation attached.
interfaceHasAnnotationWithName(Interface, AnnotName): the Interface has an annotation with the
                                                      specified name attached.
interfaceHasMethod(Interface, Method): the Interface declaration contains Method.
```

Method relations

```
methodReturnClass(Method, Class): the Method returns a result of type Class.
methodReturnInterface(Method, Interface): the Method returns a result of type Interface.
methodReturnType(Method, Type): the Method returns a result of type Type.
methodHasParameter(Method, Parameter): the Method has the specified Parameter.
methodHasAnnotation(Method, Annotation): the Method has Annotation attached.
methodHasAnnotationWithName(Method, AnnotName): the Method has an annotation with the specified
                                                name attached.
```

Method parameters are not ordered, but can be selected by their name. It is (currently) not possible to select parameters by order (e.g. the first argument of a method).

Field relations

```
fieldClass(Field, Class): the class (type) of a field - not to be confused with the Type that
                          the field is a part of (classHasField).
fieldInterface(Field, Interface): the interface (type) of a field.
fieldType(Field, Type): the type of a field
fieldHasAnnotation(Field, Annotation): the Field has Annotation attached.
```

```
fieldHasAnnotationWithName(Field, AnnotName): the Field has an annotation with the specified
                                              name attached.
```

Parameter relations

```
parameterClass(Parameter, Class): the class (type) of a parameter.
parameterInterface(Parameter, Interface): the interface (type) of a parameter.
parameterType(Parameter, Type): the type of a parameter.
parameterHasAnnotation(Parameter, Annotation): the Parameter has Annotation attached.
parameterHasAnnotationWithName(Parameter, AnnotName): the Parameter has an annotation
                                                       with the specified name attached.
```

Using these relations between program elements, it is possible to express complex queries. The next example selects all classes in the 'myapp.gui' namespace that have a 'paint' method:

```
isNamespaceWithName(NS, 'myapp.gui'),
namespaceHasClass(NS, Class),
classHasMethod(Class, Method),
isMethodWithName(Method, 'paint').
```

The first line selects only the Namespace named 'myapp.gui'. The second line states that we are interested in the Classes contained by the namespace (NS) selected in the previous part; the ',' between the two lines works like a logical 'and' - we want to select the NS with the specified name *and* also, this NS must contain Class. The third line then specifies that this Class must have Method, while the 4th line specifies that we only want to match methods named 'paint'. If the interpreter succeeds in finding one or more of such combinations, it will return all combinations of (NS, Class, Method) that satisfy these conditions, i.e. the NS is named 'myapp.gui' and contains Class, Method is named 'paint' and is contained by Class.

4.3.4 Views on the .NET model

In this section we demonstrate the power of using a predicate language by defining several views on the meta model. For example, users may want to query the 'static' aggregation structure (namespaces-classes-methods-parameters), or the inheritance structure (classes-subclasses-subsubclasses, and so on). Because we use a generic predicate language, it is possible to define recursive predicates as 'views' on the structure. These views are directly derived from the static model.

For example, we define the following aggregation view on the static structure:

Aggregation view

```
directAggregation(Parent, Child) :- isParentNamespace(Parent, Child).
directAggregation(Parent, Child) :- namespaceHasType(Parent, Child).
directAggregation(Parent, Child) :- typeHasMethod(Parent, Child).
directAggregation(Parent, Child) :- classHasField(Parent, Child).
directAggregation(Parent, Child) :- methodHasParameter(Parent,Child).

inAggregationTree(Parent, Child) :- directAggregation(Parent, Child).
inAggregationTree(Parent, Child) :-
  directAggregation(Parent, Between),
```

```
10   inAggregationTree(Between, Child).
```

The directAggregation predicate matches all the 'has' relations between program elements. The inAggregationTree predicate then enables us to walk this tree recursively - using only basic prolog programming constructs. However, we have to take care that we do not introduce the possibility of infinite loops, as is always the case when defining recursive procedures. In this case the underlying hierarchy does not contain cycles, so we are certain no infinite loops can occur.

In some cases it is desirable to select sub-elements or the element itself, so we also define the additional predicate:

```
1  inAggregationTreeOrSelf(Parent, Anychild) :-
2    inAggregationTree(Parent, AnyChild).
3  inAggregationTreeOrSelf(Self, Self).
```

Inheritance view

For inheritance we define a similar predicate, using the isSuperClass relation:

```
1  classInherits(Parent, Child) :- isSuperClass(Parent, Child).
2  classInherits(Parent, Child) :-
3    isSuperClass(Parent, Between),
4    classInherits(Between, Child).
5
6  classInheritsOrSelf(Parent, Child) :-
7    classInherits(Parent, Child).
8  classInheritsOrSelf(Self, Self).
```

If needed, it would be easy to define additional views. However, as mentioned above, great care should be taken that such recursive views do not contain loops, to ensure that the execution will terminate. Fortunately, the inheritance structure in .NET is a tree, so it does not contain any cycles.

4.4 Usage examples

Here we present a few sample queries, to give a better understanding of how the mechanism works in practice:

4.4.1 Select the class 'myapp.Collection' and all its subclasses

```
1  sel = { C |
2    isClassWithName(Collection, 'myapp.Collection'),
3    classInheritsOrSelf(Collection, C)
4  }
```

From the list of all classes found in the application, the second line selects only the one named 'myapp.Collection'. If no such class exists, the predicate fails, which means there are no results. Assuming that the specified class is found, the internal representation of the 'myapp.Collection' class becomes bound to the 'Collection' variable. There can be at most 1 result, as the name of a

class (including the namespace path) is unique. On the third line, we use the recursive definition 'classInheritsOrSelf' as defined in the previous section, to select the 'myapp.Collection' class itself, and all its subclasses. These subclasses become bound to the variable 'C'.

As there can be an arbitrary amount of variables in a predicate expression (in this case there are two: Collection and C), we need to specify which one should be considered the 'result' of this query. This is specified using the enclosing set notation on the first and last line.

It is fairly easy to convert the statement into english:

The result *assigned* ('=') to 'sel' is the *set* ('{ .. }') of program elements bound to the variable C, *where* Collection is a class named 'myapp.Collection' *and* (',') C is a class that inherits from this Collection class.

4.4.2 Select all classes in the 'myapp.gui' namespace

```
1 sel = { C |
2   isNamespaceWithName(NS, 'myapp.gui'),
3   namespaceHasClass(NS, C)
4 }
```

Structured similarly to the example above, this selector selects all classes in the namespace specified by NS.

4.4.3 Select all classes that have at least one method that returns a String

```
1 sel = { C |
2   isClassWithName(Str, 'java.lang.String'),
3   methodReturnClass(M, Str),
4   classHasMethod(C, M)
5 }
```

This example is somewhat more complex, and demonstrates the power of expressing constraints on the static structure of the program. Suppose that we want to inspect the return values of all methods that return a String - for example, because we want to convert these Strings to a different encoding as part of an internationalisation concern.

To do this, we select the 'java.lang.String' class (line 2), then look for methods that have this class as their return type (line 3), and finally look up the classes that these methods are a part of (line 4), as the Internationalisation concern has to be superimposed on these classes. Even if a single class has multiple methods that return a String, the result will contain each class only once (the result is, afterall, a *set* of program elements).

The next chapter discusses how this predicate language is integrated in Compose*.

Chapter 5

Realisation of the selector language

Simplicity does not precede complexity, but follows it.
– Alan J. Perlis, 1982

This chapter describes the implementation of the predicate-based selector language described in the previous chapter.

5.1 Choosing a logic predicate interpreter

If we want to reuse an existing predicate language interpreter, it has to meet the following practical requirements:

- It must have a bi-directional interface to Java. On one hand, we want Java classes to be able to ask the predicate engine to evaluate a predicate and get back the results. On the other hand, predicates should be able to get information from Java - the 'fact base' consisting of program elements and their relations must be made available to prolog.
- It must be available under an Open Source license; as Compose* is released under the General Public License (GPL) the interpreter needs to be available under the GPL or LGPL to avoid licensing issues. This is also convenient in case we need to make (minor) changes to the original interpreter, for example related to error reporting.

There are several alternatives that meet these requirements, most notably TyRuBa [TyR04] / JQuery [JQu03], InterProlog [Int05] and Kernel Prolog [Ker00]. We chose to use Kernel Prolog because it is written entirely in Java, is small and unoptimised (so it is easy to find out how things work internally), and it offers a powerful way of combining Java and Prolog: its 'builtin predicates' mechanism enables us to define predicates that are implemented in Java. Its biggest disadvantage is that the execution is rather slow and unoptimised, but this is acceptable to us, as it is only used at compile time. It does not slow down the modified application at runtime.

5.2 Global design

An important factor in creating a concrete design for our predicate-based selector language is the connection between Java and Prolog. In this section, we describe how this connection can be

made. Based on this knowledge, we present a global design that connects the prolog interpreter to Compose*.

5.2.1 Connecting Java to Prolog

The first part is concerned with the connection from Java to Prolog. This part is straightforward, as Kernel Prolog contains an interface object that we can send predicate queries to. The interpreter evaluates the query and returns back to Java whether it failed or succeeded. To give an example:

```
1 isNamespaceWithName(NS, 'java.util'), namespaceHasClass(NS, C).
```

This predicate query contains two unbound variables: NS and C. When we ask the interpreter to evaluate this predicate, it tries to find combinations of NS and C such that NS is a namespace with the name 'java.util' and C is a class that is part of namespace NS. If no such combination can be found, the predicate fails. If it succeeds, we obtain a list of all the combinations of values for NS and C that fulfill the constraints expressed by this query.

This part is straightforward because this is close to the standard way in which Prolog operates. The only difference with operating in Prologs standard interactive mode (where a user can type queries onto a prompt and the list of answers is presented on the console) is that we feed the queries to the interpreter through a Java interface, and also obtain the results as objects (e.g. Strings) in Java.

5.2.2 Connecting Prolog to Java

In the example above, the prolog interpreter has to know whether 'java.util' is a namespace in the current application, and what classes are inside this namespace. We have this information available in Java, in the Compose* repository, but we need a way to make this information available to the prolog interpreter. Basically, there are two ways to do this. The normal prolog way would be to define a 'fact' for every piece of information in the repository. This fact base would be generated by Compose* and loaded by the prolog interpreter before evaluating a query, so the interpreter could use the available 'facts' to resolve the query. For example, such a prolog 'fact base' representation of the repository could look like this:

```
1 isNamespaceWithName(1, 'java.util').
2 isNamespaceWithName(2, 'java.net').
3 isClassWithName(3, 'Hashtable').
4 isClassWithName(4, 'HashSet').
5 isClassWithName(5, 'Socket').
6 namespaceHasClass(1,3).
7 namespaceHasClass(1,4).
8 namespaceHasClass(2,5).
9 [..etc..]
```

When evaluating the example query mentioned in the previous section, the prolog interpreter would find that there exists a namespace with id '1' that has the name 'java.util'. Next, it would see that the classes with id 3 and 4 belong to this namespace. In Java, we could then map back the variable bindings for C (i.e. id's 3 and 4) to the real classes that are attached to those id's.

This method has a big disadvantage: it is necessary to generate a fact base representing all the knowledge about program elements and the relations between them. This fact base would be a completely redundant copy of the information that is already in the repository. This is not only a waste of resources, but also introduces problems related to keeping the redundant data stores synchronised. This is important because we want to have the ability to add relations between elements while Compose* is running.

Fortunately, Kernel Prolog offers another way to implement a connection from Prolog to Java: it is possible to define *builtin* predicates. These are simply predicates that are implemented in Java. They can be registered with the prolog engine so that it recognises them when they are used in a query. For example, by creating a builtin 'isNamespaceWithName' predicate, the interpreter would recognise this predicate on evaluation of the query and call the corresponding execution method in Java. The Java method can inspect the arguments of the predicate - in this case there are 2 arguments, both of which can be bound or unbound. If the second argument is bound (as in the example above), the Java method can look in the repository to check whether a namespace with the specified name indeed exists, and bind that namespace to the first argument.

This way, no information is duplicated: the information about program elements and relations is directly obtained from the repository in Java. However, we have to take care that the builtin predicates work in a correct and consistent manner; it is trivial to create a builtin that breaks the normally expected behaviour of a predicate (which is that it tries to find bindings for its unbound variables upon evaluation), or to implement builtins in a way that breaks the consistency of the reasoning on the prolog side of the application (e.g. by creating builtins that have side effects on parts that are used as 'facts' by other builtins).

In the next section, we present a design that uses the second alternative of communicating between Prolog and Java.

5.2.3 Design overview

Diagram 5.1 shows the global design of the Logic Language (LOLA) module and its environment.

When the Compose* compile-time is started, all concern sources (.cps files) are parsed by the Compose* Compiler (COPPER). The resulting parse tree is stored in a central repository, together with type information about the application under compilation, which is extracted by the Type Manager (TYM). When the LOLA module is started, it looks in the repository for predicate-based superimposition selectors. The main LOLA class (controller) sends each query to the prolog engine (Kernel Prolog). The interpreter resolves the queries, which will contain 'calls' to the built-in language model predicates implemented in another part of the LOLA module. These builtin predicates directly query the repository to get information about the static program structure. The results (whether the query succeeded, and when it did, all possible variable bindings) are obtained from the prolog engine by the LOLA controller part. The controller adds the results to the central repository, so that they can be used by subsequent modules.

The rest of this chapter describes this design in more detail.

5.3 The repository

The central repository is where Compose* stores all its data. Each module has inputs and outputs that are usually obtained from and stored in this repository, such that other modules

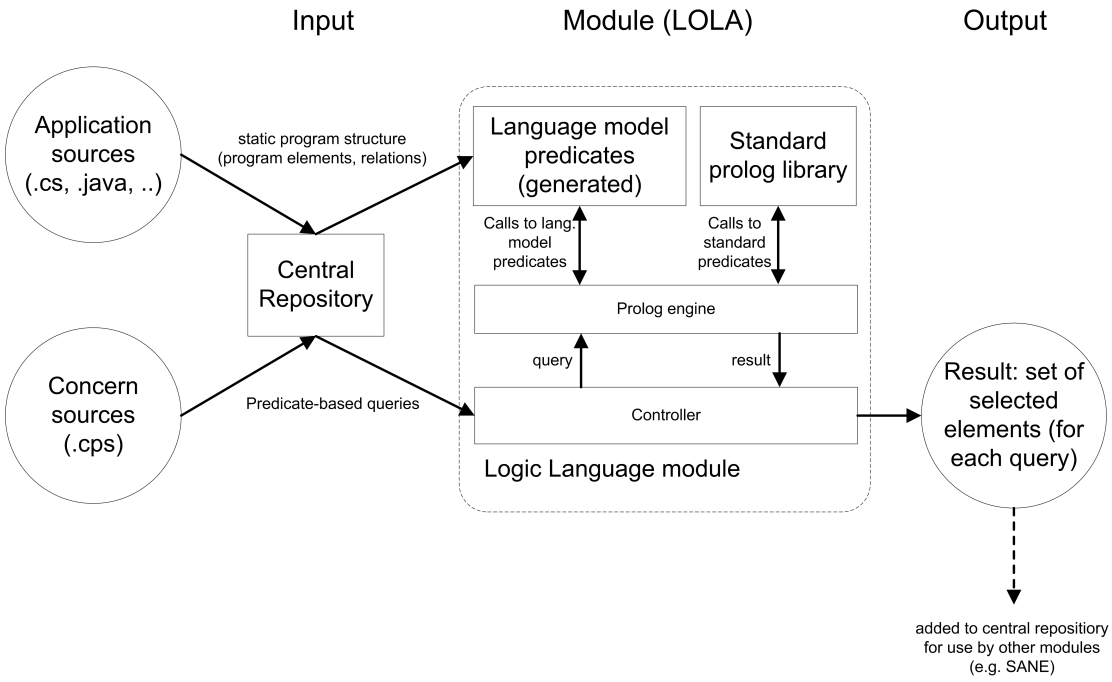

Figure 5.1: Global design

can use their results. The description in this section is limited to the inputs that are actually used by LOLA. These are: information about the static program structure and the predicate-based queries extracted from the concern sources (see figure 5.1).

The representation of the predicate-based queries in the repository is almost trivial: they are extracted from the concern sources and the entire queries are stored as objects of the PredicateSelector type in the repository. A PredicateSelector object consists of a string containing the entire query, the name of the desired result variable within the predicate, the name used to refer to this selector from within Compose*, and a reference to the superimposition part and concern that this selector belongs to.

Figure 5.2 shows the structure of the part of the repository that represents the static application structure as collected by TYM. The structure closely resembles the .NET language model (figure 4.2). The most interesting property of this structure is that the generic notion of ProgramElements (figure 4.1) is represented here as an interface that is implemented by all the classes that represent the structural elements of an application written in .NET. This allows us to access elements of any language unit type in a uniform way through the ProgramElement interface, which helps us in defining a generic interface between the Java and Prolog parts of the module.

The Type Harvester (TYM) module collects information about all the classes, methods, etc. of the application and creates DotNETType, DotNETMethodInfo, etc. objects for each of them. For example, let us take the class java.util.HashMap, which contains a method 'public Object put(Object key, Object value)'. This method would be represented as a DotNETMethodInfo object that contains the name of the method (put), keeps track of its collection of parameters (key and value), has a pointer to its containing DotNETType (in this case java.util.HashMap) and also to the return type of the method (java.lang.Object). Also, the DotNETMethodInfo object knows that this method has the 'public' attribute. Note: the class diagram in figure 5.2

only shows how the relations between program elements are stored; in reality the classes contain extra fields to store all the information about attributes, name of the program element etc.

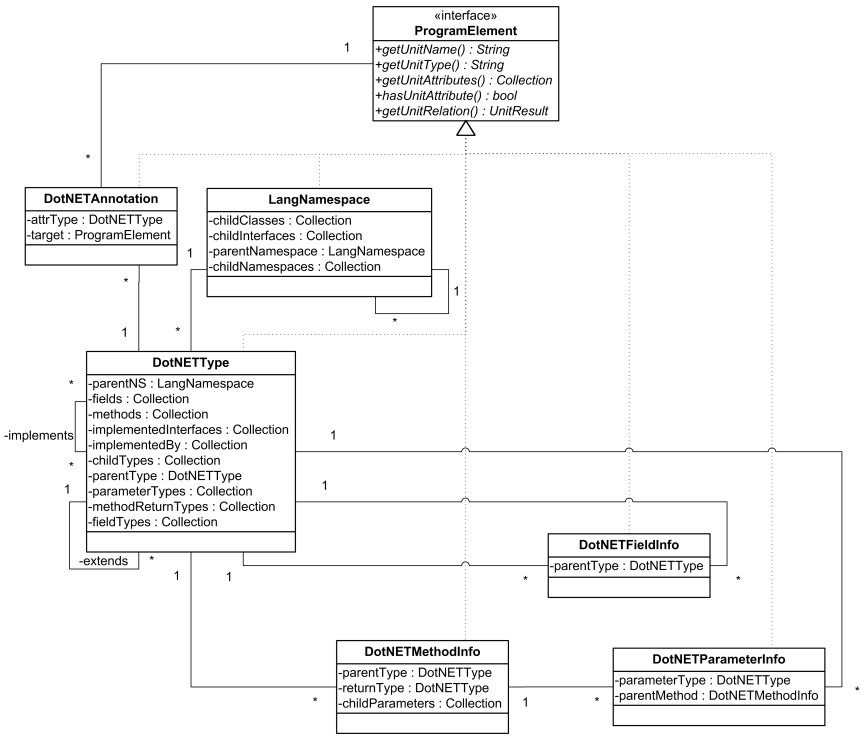

Figure 5.2: Representation of .NET static application structure in the repository

Each program element type has different kinds of relations to other program element types. Even so, we want to access them in a uniform way through the ProgramElement interface. To accomplish this, each of the classes representing a type of program element in .NET implements the five methods declared by the ProgramElement interface. Each of these methods has a specific task:

- getUnitType() returns a string containing the type of program element. In many cases, this method simply returns a fixed string (e.g. 'Method' or 'Parameter'); however it is perfectly possible to represent multiple program element types by a single class. This is what happens in the DotNETType class for example: we want to make a distinction between Interface and Class elements, but their representation in the .NET reflection framework is the same (except for the fact that an interface can not have fields). It would not make sense to split the low-level representation in separate classes (this explains the main difference between figure 5.2 and 4.2). Still, the distinction is important to us as classes can have the *implements* relation to interfaces, but obviously not the other way around.

- getUnitName() returns the name of a program element. In many cases, application programmers want to refer to specific program elements by their name, so this is an important property. Please note that the name of a unit is generally not unique, for example there can be many methods (even within the same class) that have the same name. However, we choose to represent namespaces and classes/interfaces by their fully qualified name (e.g.

java.util.Hashtable), which is guaranteed to be unique even though there can be multiple classes with the same name (to give a well-known example: java.awt.List and java.util.List) if the namespace is disregarded.

- getUnitAttributes() returns the collection of attributes attached to this program element. The kind of attributes that can be present depends on the actual program element type (which can be obtained through getUnitType). In the current implementation, only the distinction between public/private elements is implemented for the program element types to which it applies (classes, types, methods).

- hasUnitAttribute(String attr) returns true if and only if the program element has the specified attribute attached.

- getUnitRelation(String relation) defines the relations to other program elements. The caller has to specify a relation (e.g. 'ParentType') supported by this type of program element (e.g. Method). The set of supported relations to other program elements depends on the value returned by getUnitType. The getUnitRelation method returns an object of type UnitResult, which is a wrapper class that can represent either a single program element or a collection of elements - keep in mind that relations can have a 'unique' result (e.g. a Method has exactly one ParentType) as well as 'multiple' results (e.g. a Method can have an arbitrary amount of ChildParameters).

Although we have now defined a uniform interface to the program elements, the prolog engine needs to be able to tell what kinds of program elements exist, and what relations they can have to other program elements. This is why we implement an explicit higher level representation of the language model.

5.4 The language model

We want uniform access to all types of program elements, as this allows us to separate the low-level connection between Java and the Prolog engine from specific program element types. To accomplish this, we need an explicit model of the types of program elements and the possible relations between them. We implement this language model in such a way that it has an additional advantage: because we have an explicit representation of the language model, it is possible to automatically generate the predicate library that can be used by the application programmer to query the language model.

Figure 5.3 shows the design of this language model representation. A LanguageModel consists of LanguageUnitTypes (i.e. the different kinds of program elements that can be found in the language) and RelationPredicates. LanguageModel is an abstract class that defines tables containing references to all the LanguageUnitTypes and RelationPredicates for a particular concrete LanguageModel. It also implements the trivial get/set/add-methods used to reach the contents of the language model.

The DotNETLanguageModel is a concrete language model, which overrides the most important methods of the abstract LanguageModel class:

- the createMetaModel() method defines all LanguageUnitTypes, RelationTypes and RelationPredicates. The use of these classes is best explained through some examples.

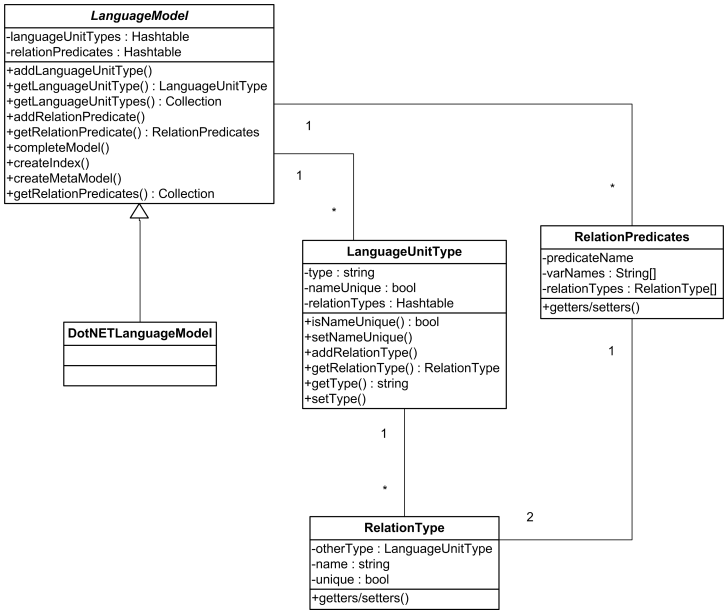

Figure 5.3: Representation of the language model

First, we define the language unit types. The example below shows the definition of the Namespace and Class language unit types. The constructor specifies the name of the unit type, (which should be exactly the same as what will be returned by the getUnitType() method of a program element of this type) and whether the name of specific program elements should be considered unique. For namespaces and classes this is the case, as we use their fully qualified names to identify them by name. The names of methods, fields and parameters are not unique within a single application, however.

```
1  // Define the 'Namespace' language unit type
2  LanguageUnitType utNamespace = new LanguageUnitType("Namespace", true /* unique name */);
3  addLanguageUnitType(utNamespace);
4
5  // Define the 'Class' language unit type
6  LanguageUnitType utClass = new LanguageUnitType("Class", true);
7  addLanguageUnitType(utClass);
```

Next, we define the possible relations to other program element types for each language unit type, and attach these so-called *relation types* to their respective LanguageUnitType. Each relation type is described by three parameters: the name of the relation (which can be used as an argument to the 'getUnitRelation()' method of a ProgramElement of this type), the unit type that can be expected on the other end of the relation, and whether this relation points to a unique element or (potentially) multiple elements of that type.

The listing below shows the definition of the namespaceChildClasses relation type (a namespace can contain multiple ChildClasses) and the classParentNamespace relation type (each class is contained by exactly one namespace). Note that each RelationType only defines one half of a relation: it specifies a one-way connection *from* one element *to* another. All relations have to be defined in both directions. This is necessary because of the way Prolog works: in a query, either side of a relation can be specified (bound) or unspecified (unbound). If we would only store the relation in one direction, looking up the relation

in the other direction would be really time-consuming[1]. For this reason, there are two RelationType objects that describe the same relation, but in opposite directions.

```
1  /** Namespace **/
2  RelationType namespaceChildClasses =
3    new RelationType("ChildClasses", utClass, RelationType.MULTIPLE);
4  utNamespace.addRelationType(namespaceChildClasses);
5
6  /** Class **/
7  RelationType classParentNamespace =
8    new RelationType("ParentNamespace", utNamespace, RelationType.UNIQUE);
9  utClass.addRelationType(classParentNamespace);
```

Finally, we define the RelationPredicates that specify the binary relations that exist in this language model. In other words, each of these RelationPredicates corresponds to a line in the language model (figure 4.2). The example below defines the relation between namespaces and classes. This relation predicate is called 'namespaceHasClass' and has two parameters, 'Namespace' and 'Class'. If the Namespace variable is specified (bound), we look at the namespaceChildClasses relation type, which (as defined above) states that we can ask the namespace program element for the units at the other end of the 'ChildClasses' relation. We also know that we should get an arbitrary amount of results (zero or more) of the 'Class' type. If, on the other hand, the "Class" variable is bound in the query, we can use the classParentNamespace relation type instead, to get from that particular Class element to its parent namespace. As specified by the classParentNamespace relation type, this will always result in a single program element of the 'Namespace' type.

```
1  /********* Definition of relation predicates *****/
2  RelationPredicate namespaceHasClass =
3  new RelationPredicate("namespaceHasClass", namespaceChildClasses, "Namespace",
4                                             classParentNamespace, "Class");
```

To summarise, a language model is represented by instances of the three classes mentioned above; the createMetaModel method is overridden by a concrete LanguageModel implementation class to create instances that represent a particular language model.

- the completeModel() method is used to ensure that the model is complete and consistent. For example, in the .NET model there exists no explicit program element in the source code representing namespaces. This method extracts the namespaces used in the application from the fully qualified names of all the classes and creates the corresponding LangNamespace (see figure 5.3) objects.

  This method also checks that all relations are actually defined in both directions (in the repository), and adds the 'backward' references if necessary.

- the createIndex() method creates a UnitDictionary, which acts like a hashtable containing all program elements in the application, with the addition of convenience methods to directly request all program elements of a certain type, all elements with a certain name, etc. The UnitDictionary keeps several indices on the program elements in the dictionary to make commonly used lookups done by the builtin predicates as fast as possible.

[1] e.g. like going backwards in a single linked list.

5.4.1 Generating the language model predicates

In addition to the language model representation classes, a class ModelGenerator is defined. Given a concrete LanguageModel (e.g. an instance of DotNETLanguageModel) it generates the predicates to query the program elements and relations in that particular language model. The LanguageModel contains all the necessary information to automatically generate the predicates as described in the analysis chapter (4.3.1). The actual language model to be used is simply an initialisation parameter of the LOLA module and could easily be made project-dependent.

The analysis chapter did not specify the implementation part (right-hand side) of the language model predicates. This brings us to the most low-level part of this design: the actual connection between Prolog and Java. The next section describes this connection, explains how the generated language predicates are attached to the Java part of the implementation, and describes in more detail how the predicates are generated based on the language model.

5.5 Putting it all together

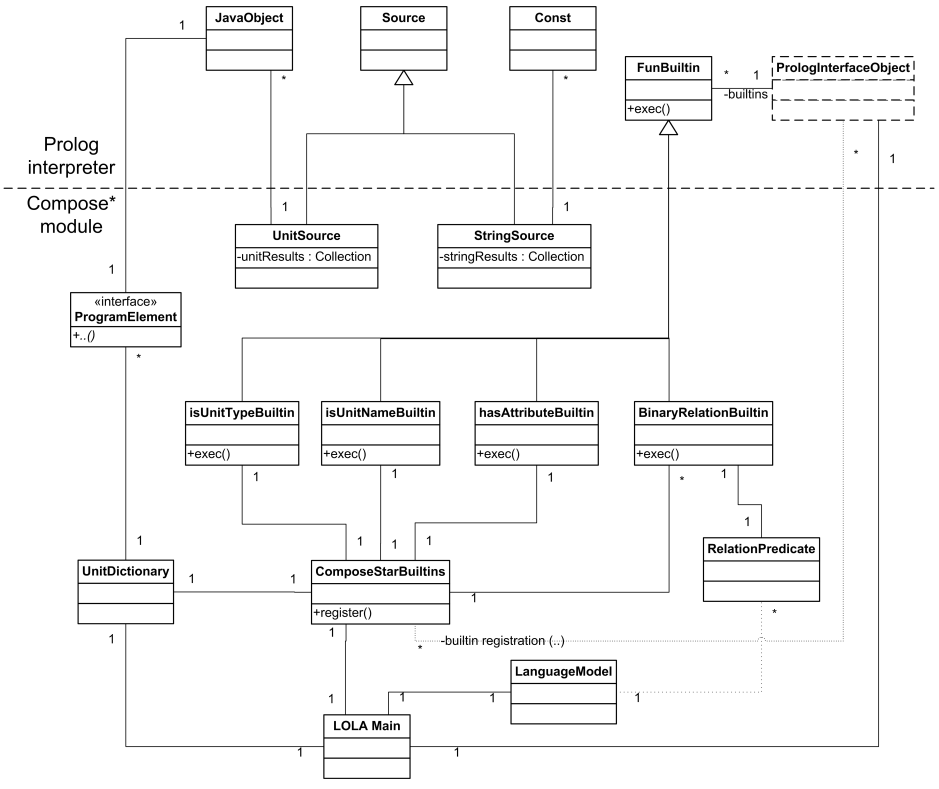

Figure 5.4: Connecting the repository to the prolog engine

Figure 5.4 shows how the prolog engine is connected to the Logic Language (LOLA) module in Compose*. The LOLA Main class is connected to a UnitDictionary, which provides quick access methods to all ProgramElements in the current application (as explained in the previous section). As described in section 5.3, these ProgramElements are actually just elements of the global Compose* repository that implement a specific interface.

The LOLA main class initialises the ComposeStarBuiltins class, which keeps a direct link to the UnitDictionary and instantiates the built-in predicate classes defined by Compose*. These built-in predicates extend the 'FunBuiltin' (built-in function predicate) class, which is part of Kernel Prolog. Each FunBuiltin instance implements a specific function predicate (i.e. a predicate with at least one argument, such as 'isClass(C)'). The instances are registered with the prolog engine, so that it will recognise the name of the predicate in queries and call the 'exec()' method of the corresponding FunBuiltin instance rather than doing the normal predicate resolution steps.

As the built-in predicate classes are the heart of the implementation, we go into some detail to explain their inner workings. Also, we give examples to demonstrate how the language model predicates are generated by the ModelGenerator, and how they are combined with the built-in predicates to implement our query language.

5.5.1 isUnitTypeBuiltin

ComposeStarBuiltins instantiates exactly one instance of the isUnitTypeBuiltin class. This instance implements the 'isUnitTypeBuiltin' predicate. To show how this built-in is used by the language model predicates, it is best to first have a look at the definition of these predicates:

```
1 isClass(ProgElem) :-
2   check_or_gen1(isUnitTypeBuiltin(ProgElem, 'Class')).
3
4 isNamespace(ProgElem) :-
5   check_or_gen1(isUnitTypeBuiltin(ProgElem, 'Namespace')).
6
7 [and so on for the other program element types]
```

As mentioned before, these language model predicates are generated from an instance of a LanguageModel (e.g. the DotNETLanguageModel) by the ModelGenerator and loaded by the prolog engine on startup (i.e. before evaluating any queries). It is trivial to generate these predicates from the LanguageUnitType instances found in the LanguageModel.

The predicates 'isClass(ProgElem)' and 'isNamespace(ProgElem)' are part of the query language. When the interpreter encounters one of those predicates in a query, it tries to unify the left-hand side of the predicate (left of the ':-' sign) with the right-hand side. In this case, the right hand side consists of a 'call' to the isUnitTypeBuiltin predicate, which is redirected to our implementation class by the prolog interpreter.

The language model predicates have a single argument (a ProgramElement), which is either bound or unbound upon evaluation of the query. The isUnitTypeBuiltin predicate has a different task depending on whether the first argument is bound or not. These are two rather different cases:

- If the first argument is bound, the exec() method in the isUnitTypeBuiltin class can cast it to a JavaObject, which is the internal wrapper for normal Java objects in kernel prolog. We can extract the actual ProgramElement from the JavaObject, as shown by the connection in figure 5.4. The builtin exec()-method then just has to check whether the specified ProgramElement is indeed of the type specified as its second argument (e.g. 'Class'), and return 'true' or 'false' based on whether or not this is the case. The actual check is now trivial: remember that each ProgramElement has the 'getUnitType' method which should return the same string value as the second argument of the isUnitTypeBuiltin predicate.

- If the first argument is unbound, the builtin predicate should generate all possible values (bindings) for the ProgElem argument. The possible bindings are all the ProgramElemens of the type specified by the second argument. As the isUnitTypeBuiltin has access to the UnitDictionary, it can simply ask for all the ProgramElements of the specified type. There is only one problem: returning them to prolog. The prolog interpreter evaluates possible variable bindings one at a time; if there are several possible bindings it starts with the first, then tries to fulfill the rest of the constraints in the query given that this specific variable is now bound to this specific value. After it is finished finding bindings for other variables it requests the next possible binding for this variable, and so on.

  Such 'multiple variable bindings' are handled in Kernel Prolog by the 'Source' class. A source is simply a Java object that generates the possible bindings for a variable. Each time the getElement() method is called on a Source, it returns the next result. The kernel prolog library defines a predicate to extract the values out of such a Source on the prolog side. Thus, we define a class 'UnitSource' which extends the kernel prolog Source class. It is initialised by a Collection of ProgramElements, and returns the next element of the collection each time getElement is called by the prolog interpreter. The isUnitTypeBuiltin predicate binds this UnitSource to the ProgramElement argument. This way, the Java side of isUnitTypeBuiltin does not (have to) know what happens with the results after it is done, but the prolog interpreter can request the next value when needed.

  This is where the check_or_gen1(..) call that is wrapped around the isUnitTypeBuiltin call comes in: obviously we do not want to see the UnitSource object as the result in Prolog, as this is a (wrapped) java-object, which is of no direct use. We want to get the values out of the Source instead of seeing the Source itself. Fortunately, we can define a generic predicate to reach this goal, as Kernel Prolog has extensive meta-programming abilities.

  The check_or_gen1(..) predicate has one argument: another function predicate with at least one argument. The implementation of the check_or_gen1(..) predicate reflects on the contents of its argument and can detect, for example, whether its first argument is bound or unbound. Based on this knowledge it predicts whether the builtin predicate will just execute a check, or generate all possible values for the first argument and return these as a Source. In the latter case, the check_or_gen1 predicate extracts the values from the Source object and actually binds the values to the first argument. Essentially, by wrapping the check_or_gen1(..) predicate around any builtin that wants to return multiple results from Java to Prolog, we hide the complex process involved to reach this goal from the user.

  The need to use *Sources* to bind multiple values to a single variable is inherent to the design of Kernel prolog with relation to the communication between Java and the prolog interpreter.

To summarize, the isUnitTypeBuiltin predicate introduces the ProgramElements into prolog. We can request all ProgramElements of a certain type, or check whether a given ProgramElement is of a certain type. The next builtin predicates, isUnitNameBuiltin and hasAttributeBuiltin, deal with the properties of program elements.

5.5.2 isUnitNameBuiltin

ComposeStarBuiltins creates exactly one instance of the isUnitNameBuiltin class. This instance implements a predicate with the same name. It is used by the language model predicates as follows:

```
isClassWithName(ProgElem, Name) :-
  isUnitNameBuiltin(ProgElem, Name, 'Class').

isMethodWithName(ProgElem, Name) :-
  check_or_gen1(isUnitNameBuiltin(ProgElem, Name, 'Method')).

[and so on for the other program element types]
```

Each of the language model predicates have 2 arguments: a program element (type: ProgElement) and its name (type: Const). Because there are 2 arguments, there are now 4 possible combinations: both arguments are bound upon calling this predicate, both are unbound, only the first is bound or only the second is bound.

- If both arguments are bound, isUnitNameBuiltin just has to check that the specified ProgElem has the name specified by Name, and that it is also of the type specified as the third argument of the builtin predicate.
- If only the second argument is bound, we need to find the program element(s) with the specified type and name. In some cases, we know (from looking at the language model) that the name of an element is unique. This is the case for e.g. Namespaces and Classes, but not for Methods. This is why the ModelGenerator class has put the check_or_gen1(..) predicate around the call to isUnitNameBuiltin in the right-hand side of isMethodWithName, but not in isClassWithName. In the latter case, there can only be one possible result, which saves a lot of overhead. The results are returned in the same way as done by the isUnitTypeBuiltin.
- If only the first argument is bound, we simply look up the name of the specified ProgElem (and also check whether it is of the type specified in the third argument of isUnitNameBuiltin). As the name of a specific ProgElem is unique by definition (each ProgramElement has exactly one name, obviously), the possibility that the second argument would return multiple values does not exist.
- In the current implementation, it is not allowed to have both arguments of a language model predicate unbound. This is purely an implementation limitation that is not too hard to remove, but could lead to slow execution times as the number of possible combinations for 2 unbound arguments can be huge.

5.5.3 hasAttributeBuiltin

ComposeStarBuiltin creates exactly one instance of the hasAttributeBuiltin class, which implements a predicate with the same name. Through this predicate, we can reach attributes of the program element such as visibility ('public', 'private') and other keywords that may have been defined by this language.

```
isClassWithAttribute(Unit, Attr) :-
  check_or_gen12(hasAttributeBuiltin(Unit, Attr, 'Class')).

isMethodWithAttribute(Unit, Attr) :-
  check_or_gen12(hasAttributeBuiltin(Unit, Attr, 'Method')).

[and so on..]
```

The different cases are basically handled in the same way as isUnitNameBuiltin does. In this case, there is a many-to-many relationship between the first and second argument though: each ProgramElement can have any amount of attributes, while each attribute (e.g. 'public') can be attached to an arbitrary amount of ProgramElements. This is why each of the calls to the hasAttributeBuiltin is wrapped by the check_or_gen12(..) predicate, which can handle multiple results (Sources) in both the first and the second argument. An attribute is represented by a String object in Java. Thus, we define the additional StringSource class, which returns Const objects (the Prolog-side representation of strings) rather than ProgramElements.

5.5.4 BinaryRelationBuiltin

All relations between program elements are binary relations. As all ProgramElements implement the same interface, the actual type of program element on both ends does not really matter to the prolog engine. However, it has to know how to navigate from the given end of a relation to the other end. Fortunately, this information is stored in the language model (see figure 5.3): each RelationPredicate represents exactly one binary relation, and it has pointers to both sides of the relation (each direction is represented by 1 RelationType object). So, ComposeStarBuiltin creates an instance of the BinaryRelationBuiltin class for each RelationPredicate found in the LanguageModel. Each instance implements a different predicate with 2 arguments. The actual name of the relation predicate is retrieved through the RelationPredicate object.

Three example definitions of relation predicates:

```
1 parameterClass(Parameter, Class) :-
2   check_or_gen1(parameterClassBuiltin(Parameter, Class)).
3
4 classHasField(Class, Field) :-
5   check_or_gen1(classHasFieldBuiltin(Field, Class)).
6
7 implements(Class, Interface) :-
8   check_or_gen12(implementsBuiltin(Class, Interface)).
```

The three builtins are implemented by different instances of the BinaryRelationBuiltin class. The actual language model predicates shown above have been generated by the ModelGenerator (like all the other language model predicates). Again, there are 4 possible combinations of unbound and bound variables, which are handled in the same way as by isUnitTypeBuiltin. For example, if the first argument is bound, the BinaryRelationPredicate asks the attached RelationPredicate for the first of its two RelationTypes. The name of this RelationType is then used to reach the other side of the relation. For example, in case of the parameterClass predicate the first RelationType would be requested from the RelationPredicate; it turns out to be named 'ParameterType'. The builtin can then call the getRelation('ParameterType') method on the Parameter element to get to the related class that represents the parameter type.

In the language model predicates above, different versions of the check_or_gen(..) predicates are used. This happens because there are 4 possible types of relations in this model:

- one-to-one: In this case no check_or_gen(..) predicate is needed. In the current DotNET-LanguageModel there are no examples of one-to-one relations.
- many-to-one: uses check_or_gen1, because the first argument can render several results if the second is bound, but the second argument will always render at most one result when

the first argument is bound. An example is the parameterClass predicate: a parameter is of a specific type, but there can be an arbitrary amount of parameters that have a specific type.

- one-to-many: essentially the same as above, which is why it also uses check_or_gen1, but swaps the arguments. The builtin knows about this (as it also has access to the LanguageModel through the RelationPredicate) and will swap them back.

- many-to-many: uses check_or_gen12 because both arguments can potentially render multiple results. An example is the 'implements' relation shown above: a class can implement an arbitrary amount of interfaces, and each interface can be implemented by an arbitrary amount of classes.

As the multiplicity of each relation is stored in the LanguageModel, the ModelGenerator and the BinaryRelationBuiltin can easily obtain this information. So fortunately, these tedious distinctions are handled automatically, as long as the multiplicity of each relation is defined correctly in the LanguageModel. The relation predicates defined in the DotNETLanguageModel reflect the multiplicity of the 'relation lines' in figure 5.2.

We have to make this distinction on the multiplicity of relations, because failing to do so (e.g. just assuming that everything may be a many-to-many relation) results in massive performance hits. As these performance hits 'stack' (i.e. combining two slow partial queries results in an exponentially slower query), we *have* to optimise to this extent in order to make the language of any practical use.

5.5.5 Implementation of the ModelGenerator class

The ModelGenerator class generates the predicates that can be used to query a particular LanguageModel. Given an instance of a LanguageModel as described in figure 5.3, it iterates over all the LanguageUnitTypes and RelationPredicates in that particular model.

For each LanguageUnitType, it generates 3 predicates. Firstly, it generates the predicate to generate all elements of a particular type, e.g. isClass(Elem), isMethod(Elem), etc. The right-hand side of these predicates always consists of the isUnitTypeBuiltin predicate. This predicate generates all elements of the specified type, or checks that a given element is of the specified type. As the type itself is an argument of the isUnitTypeBuiltin predicate, the builtin predicate does not depend on a particular set of element types.

Secondly, the ModelGenerator class generates the predicate to reach the name of a particular element type, e.g. isClassWithName(Class, Name). The LanguageUnitType object representing a specific element type knows whether elements of this type have a unique name. If so, using the check_or_gen1() predicate suffices, as the second argument is always a unique value. If the name is not unique, it puts the check_or_gen12() predicate around the call to the isUnitNameBuiltin predicate. Again, the isUnitNameBuiltin takes the actual unit type as a variable, so the builtin does not depend on a particular LanguageModel.

Thirdly, a predicate is generated to reach the attributes of a particular element type, e.g. classHasAttributes(Class, Attr). As attributes always have a many-to-many relation to program elements, the check_or_gen2() predicate is always included around a call to the hasAttribute-Builtin predicate. This predicate takes the element type as a parameter as well.

The generation of predicates representing the relations between elements is more complicated: for each RelationPredicate object found in a particular LanguageModel instance, the ModelGenerator investigates the multiplicity of both ends of the relation. Based on the information obtained through the two RelationType objects that are attached to each RelationPredicate, it knows what kind of check_or_gen() predicate should be used for each case. The RelationPredicate object also specifies the name of the builtin predicate that handles a particular relation, as well as the names of the variables in the generated predicate library.

To summarise, the LanguageModel representation described in figure 5.3 contains all the information needed to generate the predicates used to query a particular LanguageModel instance.

5.6 Example control flow

It may be hard to visualise what actually happens upon the evaluation of a selector from just looking at the static structure. Therefore, this section describes the actual program flow for a simple example query.

Take the following superimposition selector:

```
1 guiClasses = { Class | isNamespaceWithName(GuiNS, 'myapp.gui'), namespaceHasClass(GuiNS, Class) };
```

When the Compose* parser module (COPPER) encounters this source fragment in a concern specification (.cps file), it creates a PredicateSelector object that stores the entire query as a string ('isNamespaceWithName(GuiNS, 'myapp.gui'), namespaceHasClass(GuiNS, Class)'), the result variable (Class) and the name that we want to use to refer to the resulting set of selected program elements (guiClasses). This PredicateSelector object is stored in the repository as a part of the parse tree.

After the Type Manager (TYM) gathers all type information of the project under compilation and creates the repository objects representing this static structure (i.e. the DotNETTypes, DotNETMethods etc.), the LOLA main module is started.

The LOLA module first initialises the prolog engine, which involves creating the interface object and loading the default predicate libraries. Next, it creates an instance of the DotNETLanguageModel, and calls the ModelGenerator to generate the language model predicates for this particular language model. These language model predicates are then loaded from file, just like a normal predicate library. Next, the LOLA module creates instances of the ComposeStarBuiltin predicate classes and registers them with the prolog engine. Then it retrieves all PredicateSelector objects from the repository and sends them to the prolog interpreter one at a time.

When a selector query is sent to the prolog engine, it internally tries to resolve the predicate from left to right. For the example case mentioned above, the first step is that the predicate will be rewritten like this:

```
1 isUnitNameBuiltin(GuiNS, 'myapp.gui', 'Namespace'), namespaceHasClass(GuiNS, Class).
```

This happens because the implementation part (right-hand side) of 'isNamespaceWithName' looks like this:

```
1 isNamespaceWithName(ProgElem, Name) :-
2   isUnitNameBuiltin(ProgElem, Name, 'Namespace').
```

All that happens is that the left-hand side is replaced by the right-hand side of the isNamespaceWithName predicate, in the hope that this may result in a binding of the variable GuiNS.

This step is simply repeated another time: the left-hand side of isUnitNameBuiltin is replaced by its right-hand side. However, as isUnitNameBuiltin is a registered built-in predicate, its 'right-hand side' is actually a Java method, so the prolog interpreter calls the exec()-method of the isUnitNameBuiltin instance. This method observes its arguments, and discovers that the first is unbound, but the second and third are bound. The third argument states that it has to look for a program element of type 'Namespace', and the second specifies that it should be named 'myapp.gui'.

So, the exec()-method consults the UnitDictionary, asking for a unit with this type and name. As namespaces have a unique name, the UnitDictionary comes up with exactly one result, or no result in case the namespace does not exist in the current project. If the UnitDictionary does not find a result, the exec()-method returns that it has failed. The prolog engine immediately concludes that the entire query does not render any results, as it is impossible to find a valid binding for the 'GuiNS' variable. In the other case, the UnitDictionary does return the Namespace program element, and the exec()-method wraps it as a JavaObject and binds this to the first argument. It then returns control to the prolog interpreter.

The first part of the query is now handled: a binding for the GuiNS variable has been found. Let's refer to this object binding as 'JavaObject_1'.

Next, the interpreter looks at the following part of the query, as the ',' between the two parts means that both parts must be true at the same time. At this point, the remaining part of the query looks like this:

```
1 namespaceHasClass('JavaObject_1', Class).
```

Again, the query is rewritten by looking at the definition of namespaceHasClass:

```
1 namespaceHasClass(Namespace, Class) :-
2   check_or_gen1(namespaceHasClassBuiltin(Class, Namespace)).
```

So, the query can be rewritten into:

```
1 check_or_gen1(namespaceHasClassBuiltin(Class, 'JavaObject_1')).
```

As explained earlier, the check_or_gen1 predicate is used when a builtin is expected to return multiple results for the first argument. This is why the arguments have been switched: the (multiple) bindings found for Class will be returned in the first argument by the namespaceHasClassBuiltin. The check_or_gen1 predicate is a meta-predicate that basically rewrites its argument into the following:

```
1 namespaceHasClassBuiltin(ClassSource, 'JavaObject_1'), element_of(ClassSource, Class).
```

The right-hand side of the leftmost predicate is again a builtin, so the exec()-method of the BinaryRelationBuiltin that handles the 'namespaceHasClassBuiltin' predicate is called. The

BinaryRelationBuiltin instance is connected to a RelationPredicate that tells it what kind of values it can expect as its arguments. So, when it detects that the second argument is bound, it knows that it must be a Namespace program element (after unwrapping the JavaObject). The first argument is unbound, so it has to find possible bindings for this argument. From the RelationPredicate object, it learns that to reach the classes that are connected to a namespace, it has to follow the 'ChildClasses' relation of the Namespace object. By calling the method 'getRelation("ChildClasses")' on the Namespace program element specified as the second argument, it gets a set of Class elements that are contained by this namespace. The builtin knows that it should expect multiple results, as this is also specified by the RelationPredicate. Therefore, it creates a UnitSource that is initialised by this set of Class elements and returns this as the result of the first argument. Let's designate this Source as 'UnitSource_1', and the predicate can now be rewritten into:

```
1 element_of('UnitSource_1', Class).
```

The element_of predicate is part of the standard Kernel Prolog library. It has two arguments: the first is a Java object that inherits from Source, the second is an unbound variable. The element_of predicate extracts a value from the Source specified in the first argument, and binds it to the second argument. As soon as the first binding for Class is found all variables in the query are bound, which means a solution has been found. The solution is a combination of values for GuiNS and Class, and is returned by the interpreter to the LOLA controller class. LOLA stores the result for Class (as this is the result variable that it is interested in, as specified by the PredicateSelector object), and asks for the next solution. As long as the UnitSource generates new bindings for Class, more results are presented by the prolog interpreter. When the UnitSource runs out of elements to bind to the Class variable, all solutions have been found and the evaluation of this selector terminates.

The end result is that all classes in the 'myapp.gui' namespace become bound to the Class variable, which is the desired result.

The values that became bound to the Class variable are gathered by the LOLA module. Each value is an instance of a ProgramElement. In this example each value actually represents a class. However, it is also possible to select different types of elements (e.g. Methods). A reference object is created to each result value, as this is the format expected by the SANE module. Additionally, a check is added to the SANE module to make sure that filtermodules are actually superimposed on classes, and not e.g. on methods. SANE displays an error message if it detects that a filtermodule is imposed on anything else than a class.

5.7 Conclusion

We conclude that the design of the repository and the ProgramElement interface, combined with the explicit higher level (meta-)definition of the language model and the use of Kernel Prolog has the following advantages:

- Kernel Prolog offers a powerful way to combine Java and Prolog without the need to duplicate our entire Java 'fact base' (i.e. the repository); by creating a uniform interface to the repository, the 'glue' layer between Prolog and Java can be kept to a minimum[2].

[2]The classes implementing this layer, i.e. ComposeStarBuiltins, the four Builtin predicate classes and the two

- Even though we have uniform access to all program elements, it is still possible to check for the correctness of a given instance of the model, because we have an explicit representation of the possible relations between program elements and the uniqueness of element names and relations.
- Because we work on an abstract level above the actual implementation language (in our case the .NET Common Type System), it is perfectly possible to 'swap out' the language model for a completely different one. At the time of writing we have only one concrete implementation of the abstract LanguageModel class: DotNETLanguageModel. However, it would not be a problem to define e.g. a FunctionalLanguageModel that describes the common elements found in functional programming languages and the possible relations between those elements. Obviously, we would need to add different representation classes in the repository. However, as long as they implement the ProgramElement interface, there is no need to modify the lower level implementation parts of our query language. Even the different predicates that would be used in a Functional programming language model could be automatically generated by the ModelGenerator, based on the language model.
- It is easy to extend the query language: essentially the *only* necessary steps are to define additional ProgramElements in the repository and adding the meta-information about them to the LanguageModel. No low-level work is involved, as long as the information is available through reflection (TYM). Even the predicates implementing the query language are automatically generated.

The execution speed of the prolog interpreter is a minor drawback: it does not optimise query execution as some more advanced prolog engines do (e.g. by using partial evaluation). Kernel Prolog is clearly not written with execution speed in mind as a major design feature, as can be observed from the fact that it creates lots of objects[3]. This drawback is however clearly compensated by the benefits mentioned above, especially because execution speed is not of critical importance to the Compose* compile-time.

Source classes consist of less than 200 lines of code.

[3]creating objects is still one of the slowest 'basic' operations available in Java.

Chapter 6

Superimposition of annotations

Have a place for everything and keep the thing somewhere else;
this is not advice, it is merely the custom
– Mark Twain

The concept of annotations has been introduced in recent programming languages to enable the attachment of meta-data to program elements such as classes or methods. This language feature is particularly interesting because it can be used to attach additional information about an application in a way that does not have direct influence on its execution, but can be used by meta-facilities such as reflection or aspect oriented frameworks to select program elements that have specific annotations.

A concrete implementation of annotations can be found in .NET, where such annotations are called 'custom attributes' [ECM02]. Java 1.5 implements a similar feature called the Metadata Facility [Sun03].

This chapter starts with a small example that explains the use of annotations in .NET. Next, we explain how annotations can be used in Compose* selectors and why this could be useful. We take this concept to the next level by proposing a mechanism for the superimposition of annotations based on program element selectors. The last part of this chapter discusses potential problems caused by superimposing annotations and proposes a solution.

6.1 Using annotations in .NET

```
1  [Persistent]
2  class Person {
3    String name;
4    int age;
5
6    [Update] public void setAge(int a) { age = a; }
7    [Query] public int getAge() { return age; }
8
9    [Update] public void setName(String n) { name = n; }
10   [Query] public String getName() { return name; }
11 }
```

Listing 6.1: Annotations in .NET

Listing 6.1 shows a simple C# class that has annotations attached to a few program elements; in this case the class and some methods. The notation '[Persistent] class Person' specifies that the program element 'Person class' has the Persistent annotation attached. The setAge method has the Update annotation attached.

Adding these annotations to the application source as no effect on its execution whatsoever. They are pure meta-data: additional information supplied by the programmer about some property that a certain program element supposedly has.

This metadata is useful because an aspect oriented framework can look it up. That way, it can be used within concerns as a criterion to select program elements that satisfy specific conditions. This eliminates the need to explicitly enumerate a list of elements to be selected in the concern itself.

This concept is not new, of course. Other ways of including meta-data about a program in the source have existed for quite some time. For example, consider the use of naming conventions: update methods have a 'set' prefix, while query methods have a 'get' prefix. Also consider marker interfaces such as the 'Serializable' interface in Java: it does not actually declare any methods, instead it only marks an object class as being of a kind that can be serialised. However, using such conventions reduces the adaptability of applications and causes other problems, discussed in detail in [NBHA05].

For our purposes, an important feature of annotations is that they can be used by a programmer to supply information about the intended *meaning* of a part of the program. Therefore, by enabling Compose* to do selection based on annotations that are attached to certain program elements, we can take a step in the direction of superimposition based on program semantics rather than just the syntax (i.e. the static structure). The next section explains how the predicate language defined in the previous chapters supports the use of annotations in superimposition selectors.

6.2 Using annotations in Compose*

```
1
2  concern ObjectPersistence {
3    superimposition
4      selectors
5        persistentClasses = { C | classHasAnnotation(C, A),
6                                  isAnnotationWithName(A, 'Persistent') };
7    [..]
```

Listing 6.2: Using annotations in Compose*

The relation between a program element and an annotation is modelled like any other relation between two program elements. The superimposition selector in listing 6.2 selects all classes in the application that have the annotation named 'Persistent' attached.

This particular selector can be used to implement persistence in a concern separate from the rest of the code. However, the annotations themselves are still attached by writing them directly into the source of the corresponding classes. This means that persistence in reusable classes cannot easily be made application specific - if two applications want to make objects of certain (reused) classes persistent by selecting based on this annotation, they can not have different sets of persistent classes. In the next section, we try to find a solution for this problem.

6.3 Superimposition of annotations

As seen in the previous section, even the use of annotations shares one particular problem with other conventions (such as naming patterns, marker interfaces and the like) for expressing metadata: the metadata is attached to the application source in a static way. This means annotations can never be application specific. Listing 6.3 gives an example: suppose that the classes X, Y and Z are part of a class library that can is reused by applications App1 and App2. Class X has to be made persistent in App1 but not App2, class Y the other way round, while class Z has to be made persistent in both applications. This can be done by using application specific annotations (App1Persistence and App2Persistence], but the annotations have to be manually attached in the source of the classes to which they apply. This leads to unwanted coupling of applications (App1 and App2) and reusable base classes (e.g. the class library containing the classes X, Y and Z), which makes it harder to reuse and maintain that code as the annotations are scattered over the source code.

```
1 [App1Persistence] class X { ... };
2 [App2Persistence] class Y { ... };
3 [App1Persistence] [App2Persistence] class Z { ... };
```

Listing 6.3: Scattered annotations

To solve this problem, rather than 'hardcoding' the annotations in the base classes we would like to have a mechanism to specify them in a separate concern source. This way, the annotations that have to do with a particular concern (such as persistence, synchronisation, security) can all be specified in one place: the concern that they belong to. A simple extension of the superimposition mechanism used for filtermodules suffices: instead of superimposing filtermodules, we add the ability to superimpose annotations on a set of selected program elements. The selector mechanism doesn't change: one or more program elements can be selected based on their name, relations to other program elements, and so on (as described in chapter 4).

Suppose, for example, that our application has a generic DataStore class which has to be made persistent. This is handled by the ObjectPersistence concern, which also marks the object classes that should be persistent - keeping everything that has to do with persistence in one place.

```
1 concern ObjectPersistence {
2   superimposition
3     selectors
4       dataStoreClasses = { C | isClassWithName(DS, 'DataStore'),
5                                classInheritsOrSelf(DS, C) };
6     annotations
7       dataStoreClasses <- Persistent;
8 }
```

Listing 6.4: Superimposition of annotations

In listing 6.4, the DataStore class (and subclasses) will be selected by the 'dataStoreClasses' selector (line 4-5). The Persistent annotation will be attached to this set of selected classes (line 7). Another application might want to make a different set of classes persistent, which is easy because the annotation is now decoupled from the base classes that it is attached to. Also, the annotations are no longer scattered over the application source, as they are specified in one place only: the 'ObjectPersistence' concern source. This makes it easier to create more reusable components.

As it is possible to use combinations of static annotations in the (original) application source and superimposed annotations, an additional benefit is that concerns can query for annotations

that may or may not have been attached by other concerns - the concern itself does not know how the annotation got there. This enables a better separation of concerns, as concerns can now communicate indirectly through the attachment of annotations.

Another problem still exists, however: first, we removed the direct references to classes in the concern source by attaching annotations in the application source. This caused scattering of annotations, so we created a mechanism to superimpose them from within the concern. The problem is that the annotations still have to be manually attached, so the concern, once again, enumerates a list of direct references to classes - this time, to specify on which classes the annotations have to be superimposed.

This problem can be solved by allowing for the *derivation* of annotations. In many cases, we can derive whether a certain annotation should be attached based on the existence of other annotations, certain statements or structural combinations of program elements (i.e. 'software patterns'). This removes the need to manually attach annotations (either in the concern source or the source of the base application).

6.4 Derivation of annotations

In the previous two sections, we introduced the possibility to use annotations in superimposition selectors, and the possibility to 'superimpose' annotations on a set of selected program elements.

These two features can be combined to create the possibility of deriving annotations. To demonstrate what is meant by 'deriving', and to explain why it can be useful, let us look at another example.

```
concern Synchronisation {
   filtermodule SyncModule {
     [..handle synchronisation..]
   }
   superimposition {
     selectors
       updateMethods =
         { Method | methodHasAnnotationWithName(Method, 'Update') };
       queryMethods =
         { Method | methodHasAnnotationWithName(Method, 'Query') };
       syncedClasses =
         { Class | methodHasAnnotationWithName(Method, 'Synchronised'),
                   classHasMethod(Class, Method) };

     annotations
        updateMethods <- Synchronised;
        queryMethods  <- Synchronised;

     filtermodules
       syncedClasses <- SyncModule;
  }
}
```

Listing 6.5: Derivation of annotations

Listing 6.5 describes a simple synchronisation concern. The goal is to intercept all calls that touch the data inside the object and synchronise calls to these methods[1]. This concern assumes that all methods that read or write the object state have the 'Update' or 'Query' annotation

[1]Similar to attaching the 'synchronized' attribute to a method in Java. However, this way synchronisation can be application-specific and could even be implemented using different scheduling techniques, e.g. using different strategies to optimise for updates or queries.

attached in the original application source. The 'updateMethods' selector (line 7-8) selects all methods in the program that have the 'Update' annotation, while the 'queryMethods' (line 9-10) selector matches all the 'Query' methods. The 'annotations' part (line 15-17) then superimposes the Synchronised annotation on the methods selected by either of those selectors. The 'syncedClasses' selector (line 11-13) will then select all the classes that contain at least one synchronised method, and the filtermodules part of the superimposition (line 19-20) superimposes the 'SyncModule' on only those classes.

The point of this mechanism is that annotations can be derived based on a combination of other program element properties and relations, including (other) annotations. This is useful because part of the reasoning can now be done automatically instead of having to add annotations by hand. Right now, it is still necessary to mark methods as 'update' or 'query' methods by hand, but by defining additional language units in the future we should be able to write selectors based on program elements such as 'assignment statement'. This way, the 'Update' annotation could be attached automatically to methods that contain an assignment statement. Based on the notion that a method updates object state, other annotations can then be attached as well, as shown in this example.

The existence of this mechanism is a logical consequence of allowing superimposition selectors based on annotations and the attachment of annotations based on superimposition selectors. It is clear that this can cause dependencies between selectors, as the result of one selector can depend on the (non-)existence of annotations that may have been attached because of superimposition based on another selector.

In the next section, we discuss the problems that can be caused by such dependencies.

6.5 Dependency problems

It is clear that by allowing the superimposition of annotations on program elements, which may in turn be selected based on whether or not they have certain annotations attached, we introduced the possibility of dependencies between selectors and superimposition. This section describes the problems that can be caused by such dependencies, and proposes a solution.

6.5.1 Consequences of having dependencies

The biggest problem with dependencies between selectors is that the order of their evaluation may matter. In the synchronisation example above, it clearly matters whether we first evaluate updateMethods and queryMethods, then impose the Synchronised annotation and then evaluate syncedClasses, or use a different order. It is clear that the evaluation of syncedClasses depends on the execution of superimposing the Synchronised annotation on the methods selected by updateMethods and queryMethods.

In this case there is just one level of dependencies, but obviously we could add another concern that again depends on the fact whether the 'Synchronised' annotation is attached. We have to detect these dependencies so that we can evaluate the selectors and superimposition statements in the right order. But what is the *right* order? A logical guideline would be: if selector Y depends on the superimposition of annotation A on program elements selected by selector X, we have to evaluate X and superimpose A on the elements selected by X *before* evaluating Y.

The issue we need to address is that the Compose* compiler is unable to detect these depen-

dencies, as our selector language is based on a Turing-complete language. Although this allows for powerful reasoning within selectors, it makes it impossible for us to reason about the results of evaluation by looking at the syntax. To demonstrate the problem, consider this selector:

```
1 YSharesAnnotWithX = { Y | isClass(Y), hasAnnotation(Y, A),
2                           isClassWithName(X, 'X'), hasAnnotation(X, A) };
```

This selector will select all classes Y that share at least one annotation (bound to variable A) with class X. However, we do not know *which* annotation is shared by both classes - and this information is necessary to determine whether this selector could depend on one of the annotation superimpositions. It is clearly impossible to infer this from the syntax of the predicate. This case is a demonstration of the more general problem that it is impossible to infer information about the execution results (here: what values become bound to each variable) from the syntax of a statement written in a Turing-complete language.

However, we can look at the *results* of selector evaluation; if it changes after we have superimposed a certain annotation, there obviously exists a dependency between them. The reverse is not true however: if the result does not change, there might still be a dependency, it just does not occur in this specific case (combination of selector, program elements/properties/relations in the application and current state of attached annotations). In other words, by performing the evaluation of the selectors, we can observe when dependencies occur, but we cannot detect *potential* dependencies independently of a particular application.

This technique obviously does not help us in determining a correct order of executing the evaluation, as we have to evaluate the selectors and superimpose the annotations in the first place to detect whether dependencies exist.

One possible solution would be to restrict the expressiveness of our selector language, such that it is (at the very least) no longer Turing-complete. However, to make it possible to reason about the language based on the syntax of selector expressions requires quite severe restrictions on the language. An example of such an approach are Conditional Transformations [Win03] [JTr05]. We do not want to impose such restrictions on our selector language, because we see the powerful reasoning enabled by using a Turing-complete language as a major feature.

6.5.2 Towards a solution; more problems

A solution to the dependency problem might be to simply iterate over the evaluation of selectors and superimposition of annotations until a fixpoint is reached - i.e. iterating until the state (the set of selected program elements per selector) does not change between two iterations. After superimposing each annotation, the selectors should be reevaluated to reflect the changes that this may have caused.

However, the proposition of using iteration brings to light that there are more problems:

- Can we guarantee that the algorithm will terminate? Circular dependencies may cause infinite loops.
- There are different possible orders of evaluating the selectors and doing the superimposition of annotations (the order in which concerns are handled is arbitrary, and even within a single concern, superimposition specifications and selectors are supposed to be declarative and thus do not imply any ordering). Different orders may cause different selector results,

which means the specification is ambiguous.

The existence of these problems is a consequence of the expressiveness of the selector language. If the programming language used to specify the selectors has a 'not' operator, we cannot guarantee termination of a dependency-resultion algorithm. It is simple to demonstrate this:

```
1 selectors
2   noA = { C | not(classHasAnnotationWithName(C, 'A')) };
3 annotations
4   noA <- A;
```

The selector 'noA' selects all classes that do not have the annotation named 'A' attached. We then superimpose the annotation A on these classes. However, now the 'noA' selector does not match anything anymore, so the annotation should not have actually been attached! If we 'undo' the superimposition, 'noA' will again match the classes where we removed it, and thus we have created an infinite loop caused by what we will call *negative feedback* between selectors and superimposition of annotations. In this case the problem is very obvious; in reality, the selector and superimposition might have been divided over different concerns. This example demonstrates that we cannot ensure that iterating over selectors and superimposition of annotations will terminate.

To demonstrate that the order of evaluation and superimposition can make a difference, consider this example:

```
1 selectors
2   noA = { C | not(classHasAnnotationWithName(C, 'A')) };
3   noB = { C | not(classHasAnnotationWithName(C, 'B')) };
4 annotations
5   noA <- B;
6   noB <- A;
```

In this case, suppose there exists a class that has neither annotation A nor B attached - so it is selected by both 'noA' and 'noB'. If we superimpose the annotation 'B' on it (line 5), it does not match selector 'noB' anymore, so annotation A will not be attached. However, if we first superimpose the annotation A (line 6), it will not match selector 'noA' anymore, so annotation B will not be attached. So there are 2 possible endresults (either annotation A or annotation B is attached). The specification is clearly ambiguous, as there is no way to discern which order of evaluation was intended by the programmer.

Based on these examples, we can make several observations:

- An important issue is that we want selectors and superimposition to be declarative, which (among other things) means the order of attaching annotations *should not matter*. Implying an ordering endangers the declarative nature of selector specifications, which leads to problems regarding evolvability: introducing additional selectors may change the implied ordering. In addition, this makes it harder for programmers to see what is actually selected by a particular selector.

- Having to remove an annotation (as done in the first example) when it has been attached earlier is undesired: annotations can also be attached statically (in the program source), and such annotations clearly cannot be removed. On the other hand, not removing su-

perimposed annotations to reflect changed selector results also leads to very unexpected results.

- The root cause of the trouble in both cases is the same: negative feedback between selectors and superimposition of annotations causes different results based on the order of evaluation, while mixing of positive and negative dependencies can cause infinite loops.

These observations lead to our solution proposal: just disallow negative feedback between selectors and superimposition of annotations. This does not limit the expressiveness of the selector language itself: it is still possible to use 'not' and similar constructs. However, selecting based on the *absence* of a certain annotation when this annotation is attached by another concern will be considered an error, as this causes negative feedback (see the first example in this section). Adding this restriction does not only solve our practical problems (as will be demonstrated shortly), but also makes sense from a conceptual point of view: it is much easier to understand what is actually selected by certain selectors because annotations can only be attached, never removed. Additionally, we do not allow for different endresults, so that ambiguous specifications like the second example will also be considered an error case. In fact, we think that cases where the ordering makes a difference can not occur without negative feedback - but as long as we have not proven this, we check for this condition and mark such cases as ambiguous (thus 'wrong'). In short, we just ensure that the superimposition specification of annotations is actually declarative (i.e. the order of evaluation does not matter).

Although this solution is conceptually straightforward, we still have to detect cases where negative feedback or different results occur and report such cases as an error. We also have to avoid infinite loops within the dependency resolution mechanism.

In the next section we propose an algorithm that iteratively handles the superimposition of annotations while detecting problems with negative feedback. We also prove why this algorithm will always terminate.

6.5.3 Negative feedback detection algorithm

This section describes an algorithm that implements the checks for negative feedback described in the previous section.

To describe this algorithm, we first define a few terms in a more formal way:

- (Superimposition) selector: a selector expression (e.g. $S = \{C|isClass(C)\}$) that returns a set of program elements when evaluated
- Selector result: a set of program elements selected by a superimposition selector (i.e. the result of evaluating a selector)
- (Superimposition) action (e.g. $S < -A$): the act of attaching a specific annotation (A) to a set of program elements (selected by S)[2]. An annotation can only be attached once; if a program element already has the annotation A attached, it will not be attached a second time by executing a superimposition action.
- Iteration: in every iteration step, exactly 1 superimposition action is executed. All selectors are then reevaluated, giving new (possibly different) selector results.

[2]In this context superimposition is always about annotations; we obviously do not consider the superimposition of filtermodules here.

- Negative feedback: occurs when for any selector result there exists a program element that was selected in iteration i-1 but not in iteration i.
- State: the current set of selected program elements for each selector, and a list of actions executed to reach this situation.
- Endstate: a state where the execution of any superimposition action will not change any of the selector results.

Inputs, outputs, variables

To describe the algorithm, we define its inputs and outputs first.

Inputs are:

```
selectors[0..s]: an array of superimposition selectors, described by:
  selector name, predicate, result variable

action[0..n]: an array of superimposition actions, described by:
  selector_name, annotation name
```

Also, we define a 'State' container object that contains the following information:

```
Set selectorResults[0..s]: for each selector, the set of selected program elements
integer last_action: the superimposition action (0..n) that was executed to get this state
integer prev_state: a pointer to the state before last_action was executed
```

The output can be either:

- An error condition (exception thrown) when the algorithm detects that negative feedback occurred, or that there are several endstates that have different selector results.
- An array of selector results, one for each superimposition selector in the application, representing the final selector results.

Algorithm description and pseudocode

6.5.3

The algorithm basically implements a breadth-first search: given the beginstate (where no actions have been executed yet), it performs all of the possible superimposition actions and adds a new State to a list of states if an action generates selector results that differ from those in the current state *and* the new state does not already occur in the list of states. However, if any of the selector results shrinks by executing an action (i.e. it misses at least one element in the new state that was selected in the previous state) the algorithm stops, because this is an error condition (negative feedback between selectors). If executing any action in a particular state does not render different selector results, that state is marked as an 'endState'. If there are several endstates, it is checked that they all have the same selector results. After it has handled a state, the algorithm tries the next from the list until there are no states left to handle.

We chose to use a breadth-first search rather than a depth-first (recursive/backtracking) solution because of several reasons:

- The calculation of a new 'state' involves evaluating all selector predicates, which is an expensive operation. Hence, we would like to trade memory for speed to avoid doing duplicate state calculations. We can achieve this by storing states that have been handled already so we don't add the same situation a second time - a breadth-first search already keeps such a list.
- We have to consider all (different) orderings to check whether the results are the same, so it does not matter that backtracking might find the first solution faster.
- Breadth-first searching can cause problems related to state space explosion (resulting in excessive memory usage). However, we know that most actions will generally return the same selector results, even when using a different ordering, and because we check for duplicate states it will be very hard to create cases where such state space explosion actually occurs.

However, keep in mind that this is essentially a brute-force approach, so it will always be possible to construct cases that consume a lot of memory/CPU-cycles.

Listing 6.6 describes the algorithm in pseudo-code.

```
1  dependencyAlgorithm()
2  {
3    number_states = 1; // Total number of states. Increased when a new state is added
4    current_state = 0; // State that we are handling currently
5
6    // Define initial state
7    state[0].selectorResults = evaluate(selectors);
8    state[0].prev_state = -1; // no previous state
9    state[0].last_action = -1; // no action executed to reach this state
10
11   while (current_state < number_states) // more states left to handle?
12   {
13     currentIsEndState = true; // assume this is an endstate, until proven otherwise
14     for (action = 0..n)
15     {
16       setAnnotationState(state, action); // attach annotations according to current state
17
18       tempState.selectorResults = evaluate(selectors);
19       if (tempState.selectorResults != state[current_state].selectorResults)
20       { // Executing this action changed a selector result!
21         currentIsEndState = false; // So this is not an endstate
22         if (for any i in 0..s: tempState.selectorResults[i] does not contain
23             at least all elements of state[current_state].selectorResult[i])
24           throw NegativeFeedbackException; // Encountered a negative dependency!
25
26         if (for any i in 0..number_states-1:
27             tempState.selectorResults != state[i].selectorResults)
28         { // This is a new state, add it to the list of states
29           tempState.last_action = action;
30           tempState.prev_state = current_state;
31           state[number_states++] = tempState; // Add state to list
32         } // new result found
33       } // selector changed
34     } // action loop
35
36     if (currentIsEndState)
37     { // None of the actions rendered a different result => current state is an endstate
38       if (endstate == undefined // first endstate we find is always good
39           or endstate.selectorResults == tempState.selectorResults )
40         endState = state[current_state]; // found correct endstate
41       else
42         throw DifferentEndResultsException; // Different orderings are bad.
43     } // found an endstate
```

```
44
45      current_state++; // handle the next state
46    }
47    return endState.selectorResults; // return set of selected elements for each selector
48  }
```

Listing 6.6: Dependency algorithm pseudo-code

Proving the termination of the algorithm

It is not immediately clear that this algorithm will terminate in all cases, but fortunately we can prove that it will.

Non-termination could be caused by the while-loop in the algorithm. This loop has the exit condition $current\_state < number\_states$ (both values are positive integers). In each cycle, current_state is incremented. However, number_states can potentially be incremented repeatedly within a single cycle. This could cause the algorithm to never terminate. Therefore, we inspect the circumstances under which number_states is incremented. There are 3 possible cases when executing each action within a cycle:

1 An action was executed that made at least one program element disappear from a selector result set (i.e. negative feedback occurred). This is an error condition that will terminate the algorithm.

2 An action was executed that did not change any selector result. This case will be ignored, because it has been handled already. Like in case 1, number_states will not be incremented.

3 An action was executed that added at least one program element to at least one selector result. If this results in a case that has not been handled yet (the worst case), number_states is incremented.

Only in the third case is number_states incremented. In that case we are dealing with a monotonically increasing result set (over several cycles). Also, there is a finite set of program elements that can be in each selector result set (the number of program elements does not grow during the execution of this algorithm). Therefore, case 3 will eventually cease to occur, as there will simply be nothing left to add to any selector result (which means that eventually case 2 or 1 will occur).

Because current_state is increased in every cycle, and eventually no new occurrences of case 3 can be found, the algorithm will eventually terminate when current_state equals number_states.

6.6 Integrating the superimposition of annotations algorithm

The integration of the algorithm described in section 6.5.3 in the LOLA module is relatively straightforward. Rather than feeding each predicate to the interpreter only once, the LOLA controller class uses the algorithm to repeatedly evaluate the selectors. The algorithm adds temporary 'hasAnnotation' relations to the static structure representation in the repository, to model the effect of annotation actions. It then re-evaluates the selector predicates, which will automatically pick up the changed relations as the builtins draw their information directly from

the model in the repository. When it has found a valid end result, it returns this to the main LOLA module.

To summarise, integration of the superimposition algorithm does not substantially change the control flow within the LOLA module, as it can essentially be seen as part of the LOLA controller class.

6.7 Conclusion

Annotations can be used in aspect oriented frameworks to better express selection criteria: they make it possible to select elements based on their intended meaning (i.e. design information about the application) rather than only selecting based on the static structure. This removes the tight coupling between concerns and implementation classes. By extending Compose* to enable the superimposition of annotations, it is possible to separate the annotations from implementation classes, thus preventing the scattering of annotations. Also, this enables better separation between concerns, as they can select program elements based on the existence of annotations that may or may not have been attached by other concerns. The possibility to derive annotations by combining these two mechanisms (selection based on annotations and superimposition of annotations) removes the need to manually enumerate the elements that a selector should match.

The use of derivation of annotations in several concerns could make it hard for a programmer to keep track of what will match a certain selector expression. However, by ensuring the declarativeness of selector expressions and superimposition of annotations, we keep the mechanism as straightforward as possible.

-

Chapter 7

Conclusion and future work

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
– T.S. Eliot

This chapter concludes the thesis. It contains a conclusion and evaluation. It mentions related projects and possible future work based on the work in this thesis.

7.1 Conclusion and evaluation

In this thesis we designed and implemented a predicate-based selector language integrated in Compose* for the purpose of expressing superimposition selectors. The query language is based on the generic predicate logic language 'prolog', a Turing-complete language that allows for powerful reasoning within selector expressions. Unification and recursion are key features that make prolog a convenient query language, allowing for a concise and declarative specification of selectors.

The selector language defines predicates that can be used to query the elements (classes, methods, etc.) in the static structure of a target application; properties of and relations between these elements can be used as a selection criterion. The actual selector language is generated from an explicit language meta-model of the target application implementation language. In this thesis, we specifically designed a language model to represent the .NET Common Type System, but the approach is truly independent of any implementation language and can easily be extended to support different language models.

The predicate language uses an innovative way to directly query the object-based representation (in Java) of the static program structure, without the need for an explicit intermediate Prolog fact-base. This technique is based on ideas implemented in Kernel Prolog by Paul Tarau - the added value is in the design of a generic, easily reusable and extensible interface to language models represented in Java.

We investigated the use of metadata annotations within an aspect oriented framework. We implement support to use annotations as a selection criterion in superimposition selectors. Additionally, we allow annotations to be superimposed on program elements from within concerns.

A combination of these features enables the derivation of annotations. These techniques can be used to remedy the tight coupling between concerns and base classes, as well as the scattering of annotations. This should lead to better maintainability and evolvability of applications.

The combination of selection based on annotations and superimposition of annotations can lead to dependencies between selectors. The existence of dependencies implies that the order of evaluating the selectors can make a difference, which endangers the (intended) declarative nature of selectors and superimposition. Cyclic dependencies between selectors can cause the dependency resolution mechanism to never terminate. An algorithm is developed to ensure that selectors and superimposition are declarative. Cases that can cause infinite loops and cases where the order of evaluating the selectors makes a difference are detected and reported as problems. In this way, it is straightforward to see what should be selected by each selector, without the need to impose any major restrictions on the expressiveness of the selector language.

7.1.1 Evaluation

In the problem statement, several requirements have been stated. In this section we evaluate whether these requirements have been met.

Firstly, by defining a language model of the .NET Common Type System, we allow the user to query important parts of the static application structure. An abstract model defining the general properties of program elements and the relations between them is used consistently to obtain a model that can be queried in a straightforward manner. By allowing the user to query the entire static structure, it is possible to remove the tight coupling between superimposition selectors and implementation classes - direct references to classes are no longer necessary in many cases.

Secondly, by defining mechanisms for selection based on annotations as well as the superimposition of annotations, we allow for selectors based on design information rather than syntax or structure.

Finally, by reusing an existing, general-purpose predicate language (prolog), we gain the ability to do powerful reasoning within selector expressions. This allows us to express complex queries based on any number and combination of selection criteria.

We conclude that the requirements stated in the problem statement are met. The query language is implemented and integrated with Compose*.

7.2 Related work

In this section, we discuss projects and papers that are related to the work in this thesis.

7.2.1 Sally

"Sally is a general-purpose Aspect-Oriented Programming Language based on the Java programming language. Its novel combination of parameterizable introductions and advice with a logic pointcut language enables writing highly generic and reusable aspects." [Sal05]

Sally uses a logic language to describe "Parametric introductions" [HU03] in Java. It extends the pointcut mechanism by allowing pointcut definitions to contain unbound variables (parameters)

representing e.g. the target class or method. It is possible to put constraints on the values of these parameters using the logic language. The values (bindings) of the variables can be used in the implementation part of an aspect.

Compared to Compose*, Sally does not analyse the dependencies between introductions, or the possible different orderings of introductions [Win03]. Also, it does not support the use of metadata annotations.

7.2.2 LogicAJ

LogicAJ is an extension of AspectJ, adding uniform genericity and interference analysis to the language. LogicAJ partly tries to address the same issues that are described in this thesis: "A limitation of AspectJ and other aspect languages is the need to refer explicitly to the names of base program entities (types, classes, methods, etc). This makes aspect code highly vulnerable to changes in base programs. Moreover, in order to express multiple introduction or advice that are similar but not entirely identical one often needs to write code that is largely redundant and differs only in the names of referenced base entities. Such code is difficult to maintain [..]." [Log05]

LogicAJ allows the use of (meta-)variables in pointcut descriptors. Elements of the base application can be selected using a selector language based on predicate logic - the concept is similar to the predicate language introduced in this thesis. The main difference is in the expressiveness of the predicate language: LogicAJ uses the technique of Conditional Transformations to transform generic aspects (i.e. aspects containing unbound meta-variables in pointcut designators) into concrete aspects. The predicate language used to describe Conditional Transformations is strictly defined and has a limited expressiveness - it does not allow for arbitrary reasoning, as the Compose* selector language does.

The advantage of having a language with limited expressiveness is that it enables reasoning based on the syntax of selector expressions. Thus, LogicAJ can reason about dependencies and orderings of introductions between aspects based on the syntax of selectors rather than the results of their evaluation.

The problem of dependencies between concerns is discussed in detail in section 6.5 - we implemented a different solution based on the evaluation of selectors in Compose*. A strong point of the LogicAJ approach is that it can detect *potential* conflicts that may occur when an application evolves. A strong point of the mechanism used in Compose* is that it allows for arbitrary reasoning in selectors, while it still detects dependency conflicts when they actually occur in a specific application.

7.2.3 JQuery

"JQuery is a flexible, query-based source code browser, developed as an Eclipse plug-in. It is a logic (Prolog-like) query language based on TyRuBa. TyRuBa is a logic programming language implemented in Java. The JQuery query language is defined as a set of TyRuBa predicates which operate on facts generated from the Eclipse JDT's abstract syntax tree." [JQu03]

The concept of JQuery is very similar to that of our logic language. However, JQuery supports only one programming language (Java) while Compose* supports .NET, a language independent platform. Compose* takes the language independence to the next level by defining an explicit

meta-model of the implementation language. The selector mechanism now implemented in Compose* does not depend on any concrete implementation language or framework, and is not even specific to the .NET platform.

JQuery was not reused in the design of Compose* because of practical reasons: it can not be compiled for the .NET platform as it depends on a more recent version of the JDK than implemented by J#. Also, the dependence on the Eclipse model of the application syntax tree poses practical problems.

The design and use of JQuery is explained in detail in a paper about the project [JdV03].

7.3 Future work

This section describes possible future work.

7.3.1 Using predicates to describe filter matching

Within filtermodules there exists another kind of selector expression: the filter matching part, which specifies what messages should be intercepted and sent to a certain filter. In the current version of Compose*, filter matching can be done based on the name of the target class, sender class and the intended receiving method. A future work could be to express the current matching patterns in the form of predicates, unifying the two selector mechanisms currently found in Compose*.

Additionally, it would be possible to add functionality to query the values of method parameters or fields of the sender or target object. However, querying at the object level rather than static structure level requires the predicate interpreter to be moved into the runtime. Although Kernel prolog runs in the .NET runtime with very minor modifications, this would seriously slow down the application at runtime.

7.3.2 Extending the current language model

By extending the .NET language model so that it can also describe the *implementation* of methods (i.e. by adding language units such as 'assignment statement' and the like), the derivation of annotations mechanism could become even more useful. This way, it will be possible to select elements based on the observeable behaviour of methods - e.g. it will be possible to tell whether they touch instance variables. We could investigate how existing software patterns can take advantage of this mechanism.

7.3.3 Weaving annotations in the .NET assemblies

In the current version, superimposed annotations are only accessible within the Compose* framework. They can be used in selectors to enable the derivation of other annotations, but are not written back to the .NET assemblies of the actual (compiled) application. We would like to write superimposed annotations back into the .NET assemblies, so that other frameworks or even the application itself can reach the attached annotations at runtime (e.g. using reflection).

References

[AKLO01] Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. "Discussing Aspects of AOP". *Communications of the ACM*, Volume 44(issue 10), October 2001.

[Asp04] AspectJ language documentation - http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /aspectj-home/doc/progguide/starting-aspectj.html#d0e181, 2004.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using Composition Filters. *Communications of the ACM*, Volume 44(issue 10), October 2001.

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[Brn99] Paul Brna. *Prolog Programming A First Course - http://cblpc0142.leeds.ac.uk/ paul/prologbook/*. 1999.

[Car01] Patricia Salinas Caro. Adding systemic crosscutting and super-imposition to composition filters. 2001.

[CKM+99] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The amsterdam manifesto on ocl. Sep 1999.

[Cor03a] Microsoft Corporation. "Introduction to .NET, Hello World, and a Quick Look Inside the .NET Runtime". Technical report, Microsoft Corporation, apr 2003.

[Cor03b] Microsoft Corporation. ".NET Compact Framework - Technology Overview". Technical report, Microsoft Corporation, 2003.

[Cor03c] Microsoft Corporation. "Overview of the .NET Framework". Technical report, Microsoft Corporation, 2003.

[Cor03d] Microsoft Corporation. "What is the Common Language Specification". Technical report, Microsoft Corporation, 2003.

[ECM02] *C# language specification*. ECMA International, December 2002.

[EFB01] Tzilla Elard, Robert E. Filman, and Atef Bader. "Aspect-Oriented Programming". *Communications of the ACM*, Volume 44(issue 10), October 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Programming language concepts and paradigms*. Addison Wesley, 1995.

[GL03] Joseph D. Gradecki and Nicholas Lesiecki. *"Mastering AspectJ: Aspect-Oriented Programming in Java"*. Wiley, 2003.

[Gla95] M.H.J. Glandrup. Extending c++ using the concept of composition filters. Master's thesis, University of Twente, 1995.

[Gre01] Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jeffrey Palm and William G. Griswold. "An overview of AspectJ". In *Proc. of ECOOP 2001*, 2001.

[Gyb02] Kris Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Proceedings of the Second German Workshop on Aspect-Oriented Software Development, Technical Report IAI-TR-2002-1 Universitt Bonn*, 2002.

[HU03] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, March 2003.

[Int02] ECMA International. "Common Language Infrastructure (CLI)". Standard ECMA-335, ECMA International, 2002.

[Int05] InterProlog homepage - http://www.declarativa.com/InterProlog/, 2005.

[JdV03] Doug Janzen and Kris de Volder. Navigating and querying code without getting lost - http://www.cs.ubc.ca/labs/spl/projects/jquery/jquery.pdf. 2003.

[JQu03] JQuery homepage - http://www.cs.ubc.ca/labs/spl/projects/jquery/, 2003.

[JTr05] Jtransformer: Conditional transformations homepage - http://roots.iai.uni-bonn.de/research/jtransformer/cts, 2005.

[Ker00] Kernel Prolog homepage - http://www.binnetcorp.com/download/kprolog/Main.html, 2000.

[Kim02] Howard Kim. *AspectC#: An OASD implementation for C#*. PhD thesis, Trinity College Dublin, 2002.

[KLM+97] Gregor Kiczales, Joghn Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. proceedings of the european conference on object oriented programming (ecoop). 1997.

[Koo95] Piet S. Koopmans. On the definition and implementation of the Sina/ST language. Master's thesis, University of Twente, 1995.

[Log05] Logicaj homepage - http://roots.iai.uni-bonn.de/research/logicaj/, 2005.

[NB04] István Nagy and Lodewijk Bergmans. Towards semantic composition in aspect-oriented programming. In *1st European Interactive Workshop on Aspects in Software*, Mar 2004.

[NBG+05] Istvan Nagy, Lodewijk Bergmans, Gurcan Gulesir, Pascal Durr, and Mehmet Aksit. Generic, property based queries for evolvable weaving specifications. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop*, 2005.

[NBHA05] Istvàn Nagy, Lodewijk Bergmans, Wilke Havinga, and Mehmet Aksit. Applying semantic properties in aspect-oriented programming. 2005.

[OMG03] OMG (Object Management Group). UML 2.0 OCL Specification - http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.

[OT01] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and the Hyperspace Approach". In *Software Architectures and Component Technology: The State of the Art in Research and Practise*. Kluwer Academic Publishers, 2001.

[PGA02] Andrei Popovice, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. 2002.

[PGA03] Andrei Popovice, Thomas Gross, and Gustavo Alonso. Just-in-time aspects: Efficient dynamic weaving for java. 2003.

[Sal05] Sally homepage - http://dawis.informatik.uni-essen.de/site/site/research/aosd/sally, 2005.

[Stu02] David Stutz. The Microsoft Shared Source CLI Implementation. 2002.

[Sun03] *JSR 175 Public Draft Specification: A metadata facility for the Java Programming Language*. Sun Microsystems, Inc., 2002-2003.

[TBG03] Tom Tourwe, Johan Brichau, and Kris Gybels. On the existence of the aosd-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[TO00] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*, 2000.

[TyR04] TyRuBa Homepage - http://tyruba.sourceforge.net/, 2004.

[Wat90] David A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[Wat00] Damien Watkins. "Handling Language Interoperability with the Microsoft .NET Framework". Technical report, Monash Univeristy, oct 2000.

[Wic99] Johannes Cornelis Wichman. Composej the development of a preprocessor to facilitate composition filters in the java language. 1999.

[Wik05] Wikipedia. Structured query language (sql) - http://en.wikipedia.org/wiki/sql. 2005.

[Win03] Tobias Windeln. Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.