Diploma Thesis

Complex Composition Operators for Semantic Web Languages

submitted by

Jendrik Johannes

born 03-26-1981 in Uelzen

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: MSc. Jakob Henriksson
Professor: Dr. rer. nat. habil. Uwe Aßmann

Submitted November 28, 2006

Contents

List of Figures

List of Listings

1 Introduction

Descriptions of data in the Semantic Web [BLHL01] easily grow larger and more complex than the descriptions of traditional web pages and are thus more difficult to handle. In the traditional web, data is enriched with formatting information to be presented to the human reader. The Semantic Web calls for additional machine-readable semantic descriptions to enable computational agents to support a user in acquiring the desired data from the mass of available information [SWM04]. This poses new challenges to data maintainers who have to maintain an overview of complex data descriptions. They should be supported by tools to structure such complex sets of data in the most appropriate way for them and the data at hand.

If large data repositories exist, it is unavoidable to apply queries and transformations to extract or transform the data. Such queries and transformations can grow large and complex as well. Here, tool support for structuring and reuse is also necessary to support query authors.

The problem of how to structure complex systems has been addressed in traditional software engineering for decades and is commonly solved by splitting systems into components using different techniques [McI68, Szy98, Aßm03]. While the techniques of the object-oriented paradigm — object interaction and class inheritance — are the most prominent to structure a software system, they have proven to be insufficient for many structuring needs. Thus, new techniques, like aspect-oriented programming [KLM+97], were developed.

This shift to new composition techniques in software development changed the level of component reuse from solely *black-box* to also include *gray-box* components. Traditional compiled objects are black-box components and they are not changed at composition time. In contrast, aspect-oriented programming systems (e.g., AspectJ [CC03]) change internal parts of a component upon composition, working at source code level. Such source code components reveal information about their interior (like white-box components) while maintaining a clearly defined interface for composition (like black-box components). Source code in the domain of programming languages is comparable to ontologies or queries written in Semantic Web languages, which are usually not compiled down to a binary form. Consequently, gray-box is a desired level of reuse in ontology and query composition. This in turn leads to the assumption that gray-box based composition approaches are applicable for Semantic Web components.

Compositions are executed by *composition operators*, also called *composers*. Every tool or system that composes something includes composers, either explicitly defined or hidden in the systems internals. In this thesis, we examine what kind of composition

operators can be developed to structure and reuse queries and ontologies. For this, we identify composition operators applied in software engineering and investigate how they can be transferred to the Semantic Web domain.

As a foundation to define composition operators, we utilize *Invasive Software Composition* (ISC) [Aßm03, S+04]. It is a gray-box composition approach that includes a set of *primitive composers*. Here, we investigate how these can be combined to *complex composers*.

The ideas of invasive software composition are implemented in the tool E-CoMoGen[1] [AJH+06]. It realizes the primitive composers in a language independent manner, meaning that new languages can be integrated and extended to support composition techniques executed by the primitive composers. Consequently, the integration of an ontology and a query language into the tool is possible.

We investigate how the tool can be extended to support the definition of complex composition operators. Then we define such operators for an ontology and a query language. By doing this, we are able to check to which degree complex composition operators can be language independent — i.e., can be applied for compositions in programming, ontology, and query languages alike.

1.1 What this Thesis is about

The E-CoMoGen tool, while being able to execute simple compositions for any language with the integrated primitive composers, does not yet provide a convenient way to define additional complex composers. Therefore, the first goal of this thesis is to enhance the tool architecture to support complex composer definitions. To achieve this, a suitable technique to define complex composers is identified and the result is integrated into the tool.

As mentioned, different composition approaches and techniques exist in software engineering. We can learn and get inspiration from the existing ideas to develop composition operators for Semantic Web languages. Consequently, we analyze some of these approaches, determine their key properties, and categorize them. From this analysis, we draw conclusions about possible complex Semantic Web composers.

With the prerequisites performed, the enhanced tool and the knowledge gained from existing composition approaches are used to experiment with complex composition operators for Semantic Web languages. The investigations concentrate on one ontology language — OWL [SWM04] — and one query language — Xcerpt [SB04]. One complex composition idea for each language is examined in detail.

[1] EMF based component model generator

1.2 What this Thesis is not about

The topic of complex composition operators for Semantic Web languages is broad. There might be hundreds of useful composers out there to be discovered. This thesis does not provide a complete survey of useful and useless composition ideas for Semantic Web languages. Such a survey is future work that should apply the results of this thesis.

The composition tool is extended in this thesis to meet the needs for complex composer construction. To realize this extension, other functionalities of the tool need to be altered and enhanced. Describing these modifications is out of scope of this thesis. They are documented in the tool's documentation and shall support further development of E-CoMoGen into a stable language independent composition tool.

1.3 Document Structure

In this thesis, we introduce the ideas of invasive software composition, show how different software composition approaches can be unified using these ideas, and explain how this unification helps to reuse ideas for Semantic Web composition operators.

In Chapter 2, software composition is unified using the idea of *composition systems*, and *invasive composition systems* are introduced as a means to realize different composition approaches in a uniform way. In Chapter 3, some composition approaches in software engineering are examined and put in relation to invasive composition systems. The problem to realize these approaches as complex composition operators is discussed and a solution and its implementation are proposed. The results from Chapter 2 and 3 are used to define complex composition operators for ontology components in Chapter 4 and for query components in Chapter 5. Finally, Chapter 6 concludes this thesis, summarizing the results and proposing directions of future work.

2 Composition

Composing software from components using different approaches has been studied in the field of software engineering intensively. While many well-known component architectures, like CORBA [Sie98] or EJB [Jav00], are specifically designed to deal with executable software components, other more general approaches can be transferred to other domains, like the *Semantic Web*. These approaches include the well-known composition ideas of object linking and class inheritance found in object-oriented languages, as well as many *gray-box* composition approaches.

Gray-box composition combines black-box and white-box composition. Components in black-box composition have a well-defined composition interface, while they completely hide their internals. In white-box composition, the internal parts of a component are accessible and modifiable, while a well-defined composition interface is non-existent. Gray-box composition combines the ability to modify a component upon composition with the leverage of having well-defined composition interfaces.

Following the requirement to change a component's internals, most gray-box compositions in software engineering work on source code. This corresponds to an abstraction level desired in the Semantic Web domain — ontologies, rules, or queries are seldom compiled but rather processed directly in their source format.

A generic approach of realizing gray-box composition is Invasive Software Composition (ISC) [Aßm03, AJHS06]. It introduces the flexible but well-defined concept of *invasive composition systems*, which is a specialization of the *composition system* concept. An invasive composition system consist of a *fragment component model*, which can be automatically derived from languages targeted to write components [S$^+$04], a *composition language* and a set of *composers*.

In Section 2.1, the structure and parts of *composition systems* are defined and explained on examples from software engineering. The concept of *invasive composition systems* (following the ISC approach) and how such systems can be generated automatically from a language description is explained in Sections 2.2 and 2.3. An implementation of these ideas is introduced in Section 2.4. Section 2.5 concludes this chapter with references to related work.

2.1 Composition Systems

According to [Aßm03], a *composition system* is a triple consisting of a *component model*, a *composition technique* and a *composition language*.

- **Component model:** The component model of a composition system describes what components look like. It defines the type of components that may exist in the composition system and the language they are written in — the *component language*. Most importantly, it describes what kind of composition interface components have.

- **Composition technique:** When components are composed, they are connected using a certain composition technique. A composition technique can be, for instance, library linking or the rewriting of several source code fragments into one piece of code. Composition techniques are implemented by *composition operators* or *composers*.

- **Composition language:** *Composition languages* are used to write *composition programs*[1]. Composition programs define, which composition operators are executed on which components. They describe a system as a composition of components.

2.1.1 Java's Composition Systems

While the final aim of this thesis is to develop composition operators for Semantic Web languages, the composition concepts applied are explained on composition systems that exist in software engineering. This eases the understanding of the composition concepts and stresses their language and domain independence.

One of the best known languages in software engineering is the object-oriented programming language Java. To start with, we look at the composition systems that exist in Java and identify the component models, the composition techniques, and the composition languages. We also look at the composition operators that exist in Java.

The first composition system found in Java, illustrated in Figure 2.1, enables composition through importation and inheritance. The components are classes (and interfaces). The component model says that classes can reuse other classes by importing them. Those classes can be varied and extended by inheriting from them. In inheritance, a class automatically contains all public and protected methods of a superclass and can redefine existing methods by specifying a method with the same name (and signature). A class can extend a superclass by defining methods with names (and signatures) that do not exist in the superclass.

[1] also called *composition recipes*

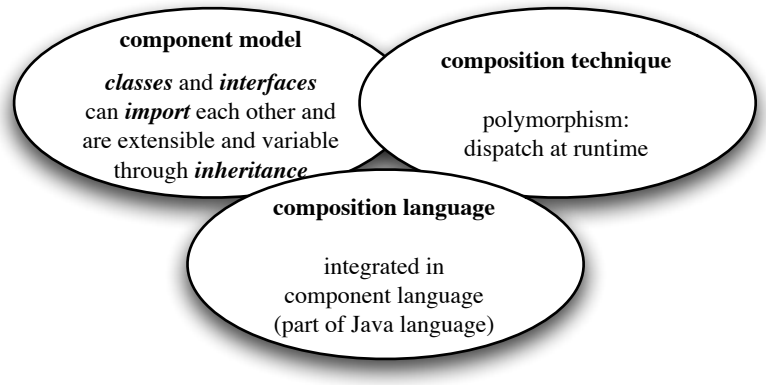

Figure 2.1: Static composition system in Java

There is no explicit composition language to describe the import or inheritance relationships. That is done in Java itself (the component language), meaning that the composition language is contained in the component language.

While the import and inheritance relationships are statically defined and do usually not change at runtime, the composition technique is to resolve them at runtime. This is due to the fact that each class can be compiled individually. The import and inheritance composition operators of Java can thus be found inside the runtime environment.

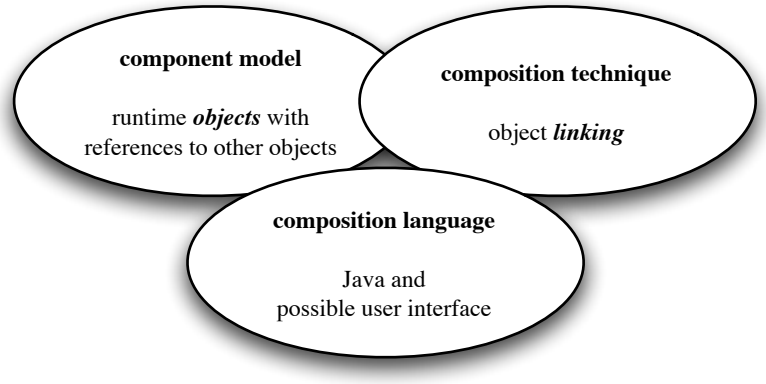

Figure 2.2: Dynamic composition system in Java

Figure 2.2 shows the second composition system found in Java, which is dynamic and applies at runtime of a program. The components here are objects. Objects contain references to other objects. These references can be changed and, if of arbitrary multiplicity, extended. The dynamics of how the objects are composed is again described in

Java itself. However, if the running program offers an interface for user input, this can be seen as part of the composition language for Java objects.

The composition technique is to link objects dynamically. The corresponding composition operator is again contained in the runtime environment.

2.1.2 AspectJ's Composition System

The composition systems presented in the last section are natively included in the Java language and runtime environment. It is also possible, and one of the fundamental ideas of invasive composition, to build a composition system on top of an existing language and an existing runtime environment. Therefore, the language might be extended and additional tooling to process compositions might be provided.

An example for how this is done, with Java and its runtime environment as a foundation, is AspectJ [CC03]. It is a composition system (Figure 2.3) that implements the *aspect-oriented* composition approach [KLM+97] for Java.

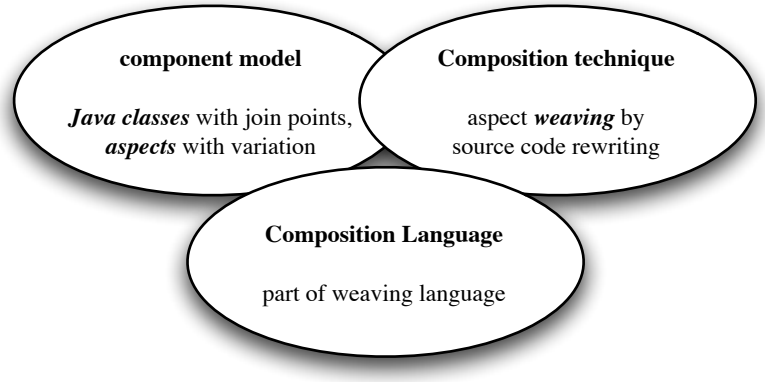

Figure 2.3: AspectJ as composition system

Aspects are always woven into a core. Thus, the component model of AspectJ allows to define cores, which are Java classes, and aspects, which are defined in AspectJ's *weaving language*[2]. AspectJ extends the semantics of the Java language in the sense that classes now can be examined for so-called *join points*. A join point indicates a position in a core where aspects may be woven. A set of join points is defined by a *pointcut* and the aspect that is woven is defined by an *advice*, both written in the weaving language. Consequently, the weaving language of AspectJ integrates parts of the component language (i.e., the aspect definitions) into the composition language (i.e., the join point selection). The aspect-oriented approach is also examined in Section 3.1.1.

[2]The weaving language of AspectJ is called *pointcut language*.

The composition technique of AspectJ is code rewriting[3]. The aspects' code is combined with the source code of the core classes and printed back to be compiled like any ordinary Java class. The corresponding composer is implemented in Java and acts as a pre-processor, passing the composed code to the native Java tooling.

2.2 Fragment Components

More specific than the definition of a composition system is the one of an *invasive composition system* in [Aßm03, p. 119]. It states that an *invasive compostion system* is a triple consisting of a *fragment component model*, a set of *composition operators* and a *composition language*.

The last section demonstrated that different kinds of component models, flavors of composition languages, composition techniques and, consequentially, composition operators exist in well-known composition systems. In contrast to specific component models, *invasive composition systems* always contain *fragment component models* (based on *fragment components*) that can be derived automatically from language descriptions and are not specifically designed for one composition approach. In contradiction to the composers identified in Java and AspectJ, which are specifically designed for the corresponding composition systems and fall into the category of complex composers (Chapter 3), Section 2.3 introduces *primitive composition operators* that work in any invasive composition system. As long as two invasive composition systems use the same (primitive) composers, they may also contain the same composition language. Such a common composition language is defined in Section 2.3.

The primitive composers, the common composition language, and the derived component model form an invasive composition system that implicitly exists for any formal language. This unification reveals a first relationship between component based development in the software engineering and the Semantic Web domain.

The fragment component model contained in an invasive composition system for some formal language is derived from a description of that language. The language description can, for instance, be a grammar that defines the constructs assembling the language by a set of grammar rules. A valid program written in the language is an instance of this grammar (i.e., a set of sentences conforming to the grammar). Parts of such a program, on their own not valid as a stand-alone program, are still partial instances of the grammar. Such a *fragment* of program code is a valid component in the component model and is called *fragment component*. A fragment component is an instance of a language construct. This construct is referred to as the fragment's *type*.

[3]As a matter of fact, different implementations of AspectJ rewrite code on different levels (i.e., one works on source code, one on Java bytecode). However, all of them rewrite code to valid Java programs.

We illustrate the idea of fragment components on Java. Listing 2.1 shows an excerpt from a grammar of the Java language[4] and Listing 2.2 shows a Java class conforming to the grammar. The Java class as a whole is a valid program[5] while the attributes and methods it contains are not. However, a method on its own (as shown in Listing 2.3) is an instance of the `MethodDeclaration` language construct. Line 7 in Listing 2.1 is the starting rule of the rule set describing the construct. It is a valid fragment component of the type `MethodDeclaration`.

```
1 ClassDeclaration     ::= Modifier Identifier "{" MemberDeclaration* "}";
2
3 MemberDeclaration    ::= AttributeDeclaration | MethodDeclaration;
4
5 AttributeDeclaration ::= Modifier Type Identifier ("=" Value)? ";";
6
7 MethodDeclaration    ::= Modifier Type Identifier "(" Argument* ")" "{" Statement* "}" ";";
8
9 ...
```

Listing 2.1: Excerpt from the Java grammar

```
1 public class Car {
2    String name = "myCar";
3
4    public void drive() {
5       System.out.println("driving around...");
6    }
7 }
```

Listing 2.2: Java class: a valid program

```
1 public void drive() {
2    System.out.println("driving around...");
3 }
```

Listing 2.3: Java method: a valid fragment

2.2.1 Genericity

Under special circumstances, fragment components can be partially specified but still remain valid. To allow genericity, they may contain "holes" where other fragment components are inserted upon composition. Such a "hole" is called *slot* and the insertion of a fragment component in its place is referred to as "*binding* the slot". Every slot has a type, depending on its position, and only accepts correctly typed fragment components for binding. A slot is part of the *declared composition interface* of a fragment component and a composition program can refer to it by its name.

[4]The presented incomplete grammar should be assumed to describe a complete language in the context of the example (Appendix B.7 contains the full grammar).

[5] More general, a Java *compilation unit* is a valid program. Such can be a class but may additionally contain *package* and *import* declarations.

The altered example in Listing 2.4 replaces the method from Listing 2.2 with a slot in the class body. The slot's name is `mySlot` and its type is `MemberDeclaration`. It can be bound, for example, with the fragment component from Listing 2.3 as their types (i.e., `MemberDeclaration`) match.

```
1 public class Car {
2    String name = "myCar";
3
4    <<SLOT mySlot : MemberDeclaration>>
5 }
```

Listing 2.4: Java class containing a slot for a `Declaration`

2.2.2 Extensibility

Not only genericity but also extensibility is supported in fragment components. Extensibility is always possible at points where the language allows, by definition, a collection of the same type of fragments. New fragment components can be inserted at any point in such collections (e.g., at the beginning or at the end). These points are called *hooks*, which can be *extended* during composition. Hooks are typed in the same way as slots. Unlike slots, which are always *declared*, hooks may also exist *implicitly* (e.g., at the beginning or at the end of a fragment list). Declared hooks are part of the *declared composition interface*, while implicit ones belong to the *default composition interface* of a fragment component.

We could change the declaration of the slot to that of a declared hook in the example (Line 4 in Listing 2.4), which can then be extended by arbitrarily many fragment components of the type `MemberDeclaration`. Implicit hooks might also exist in the list of `MemberDeclaration`s that exists in a `ClassDeclaration`.

2.2.3 Language Extensions

Since the construct used in Listing 2.4 to declare a slot is not available in the original language (Listing 2.1), a language extension is required to build a composition environment with tool support. Extensions are only needed to define fragment components and to compose them, not to define actual program logic. The handling of modularization and composition is separated from the interpretation of programs. Thus, we distinguish between the *core language* — a language targeted to write fragment components and programs — and the corresponding *reuse language* — an extension of the core language that contains constructs to handle genericity and extensibility in fragment components.

The extensions for different reuse languages follow a similar pattern: introduce a construct for slot (or hook) declarations of a certain type as an alternative to the construct representing the type. As an example, Listing 2.5 shows a grammar for a reuse language with Java (Listing 2.1) as its core language. The fragment in Listing 2.4 is an instance

of this grammar. The slot declared there instantiates the `MemberDeclarationSlot` construct.

The extension to a reuse language can be automated to achieve *universal genericity* and *universal extensibility* addressed in the next section.

```
1  ClassDeclaration      ::= Modifier Identifier "{" MemberDeclaration* "}";
2
3  MemberDeclaration     ::= AttributeDeclaration | MethodDeclaration |
4                            MemberDeclarationSlot | MemberDeclarationHook;
5
6  AttributeDeclaration  ::= Modifier Type Identifier ("=" Value)? ";";
7
8  MethodDeclaration     ::=  Modifier Type Identifier "(" Argument* ")" "{" Statement* "}" ";";
9
10 MemberDeclarationSlot ::= "<<SLOT" Identifier ":" TypeIdentifier ">>";
11 MemberDeclarationHook ::= "<+HOOK" Identifier ":" TypeIdentifier "+>";
12
13 ...
```

Listing 2.5: Grammar of a reuse language with Java (Listing 2.1) as its core language

2.2.4 Universal Genericity and Extensibility

If programs and components written in some language may contain slots as alternatives to fragments of a certain type, the language is *generic* with respect to that type. In order to achieve *universal genericity*, the language has to support genericity for every type of fragment. This means that a language description has to include slot declarations as alternatives for all constructs contained in the language.

Adding support for universal genericity to an existing language by means of defining a reuse language can be automated. The following informal algorithm derives a reuse language description from a given language description to realize universal genericity support for that language:

- For each language construct named `<X>`:

1. create a new construct `<X>Slot` that can be used to declare a slot of type `<X>`.
2. create a new construct `Abstract<X>` that references `<X>` and `<X>Slot` as alternatives.
3. replace all other references to `<X>` by references to `Abstract<X>`.

When looking at extensibility, the equivalent of universal genericity is *universal extensibility*. In the last section it was already indicated that only fragments conforming to collection like constructs are extendable. If all such fragments are extendable, universal extensibility is supported by the corresponding language.

Universal extensibility support can be achieved in different ways. If the composition language offers the ability to address implicit hooks, such implicit hooks can be assumed at any position in every collection of fragments. Universal support for declared hooks can

again be derived automatically for a reuse language. The following informal algorithm applies:

- For each reference in a construct of arbitrary multiplicity (*) to a construct <X>:

1. create a construct <X>Hook that can be used to declare a hook of type <X> if it does not already exist.
2. create a construct <X>CollectionElement that references <X> and <X>Hook as alternatives if it does not exist already.
3. replace the reference to <X> by a reference to <X>CollectionElement.

The result of the application of the algorithms to the Java grammar from Listing 2.1 is presented in Listing 2.6. Three rules (MemberDeclarationSlot, AttributeDeclarationSlot, and MethodDeclarationSlot) are introduced to express three different types of slots. In comparison, the manual extension in Listing 2.5 includes only one slot declaration rule (MemberDeclarationSlot) that is defined as direct alternative for other MemberDeclarations (i.e., without introducing an AbstractMemberDeclaration). This shows that the very generic algorithms presented above may produce a certain overhead of slot and hook declaration constructs. Thus, it is convenient to offer the possibility to tailor language extensions for specific needs. Additionally, the concrete syntax in the added rules needs to be configurable to avoid conflicts with the original language syntax.

```
1
2  ClassDeclaration      ::= AbstractModifier AbstractIdentifier "{" MemberDeclarationCollectionElement* "}";
3  ClassDeclarationSlot      ::= "<<SLOT" AbstractIdentifier ":" "ClassDeclaration" ">>";
4  AbstractClassDeclaration  ::= ClassDeclaration | ClassDeclarationSlot;
5
6  MemberDeclaration         ::= AbstractAttributeDeclaration | AbstractMethodDeclaration;
7  MemberDeclarationSlot     ::= "<<SLOT" AbstractIdentifier ":" "MemberDeclaration" ">>";
8  AbstractMemberDeclaration ::= MemberDeclaration | MemberDeclarationSlot;
9  MemberDeclarationHook     ::= "<+HOOK" AbstractIdentifier ":" "AbstractMemberDeclaration" "+>";
10 MemberDeclarationCollectionElement ::= AbstractMemberDeclaration | MemberDeclarationHook;
11
12 AttributeDeclaration      ::= AbstractModifier AbstractType AbstractIdentifier ("=" AbstractValue)? ";";
13 AttributeDeclarationSlot  ::= "<<SLOT" AbstractIdentifier ":" "AttributeDeclaration" ">>";
14 AbstractAttributeDeclaration ::= AttributeDeclaration | AttributeDeclarationSlot;
15
16 MethodDeclaration         ::= AbstractModifier AbstractType AbstractIdentifier
17                                "(" ArgumentCollectionElement* ")" "{" StatementCollectionElement* "}" ";";
18 MethodDeclarationSlot     ::= "<<SLOT" AbstractIdentifier ":" "MethodDeclaration" ">>";
19 AbstractMethodDeclaration ::= MethodDeclaration | MethodDeclarationSlot;
20
21 ...
```

Listing 2.6: Automatically derived grammar of a reuse language based on Java

2.3 Primitive Composition Operators and Composition Languages

In this section, we introduce what we refer to as *primitive composition operators* (in contrast to *complex composition operators* treated in Chapter 3). The primitive composition operators are the *Bind* and *Extend* composers, the functionalities of which have been hinted at in the last section.

The advantage of primitive composition operators is that they are applicable to *fragment components* written in any formal language working on its derived component model. Thus, they can be realized in a language independent way and executed using a common composition language.

2.3.1 Primitive Composition Operators

The names of the two primitive composers of invasive composition — *Bind* and *Extend* — indicate their functionality. The Bind composer replaces (binds) one thing with another and is thus applicable to resolve genericity. The Extend composer extends one thing with another and is utilized for extensibility.

When a composer is *called* (i.e., executed), one has to specify arguments. These arguments are one or two fragments and a slot (for Bind) or a hook (for Extend). Depending on the given arguments, the composer works in one of three different *modes*.

- **Mode one — application on variation point:** Two fragments and a slot (a hook) are given as arguments. The primitive composer addresses the slot (the hook) in the first fragment and binds (extends) it with the second fragment.
- **Mode two — application on fragment:** Two fragments are given as arguments. In the case of Bind, the first fragment is replaced by the second fragment. In the case of Extend, the second fragment is added to the collection containing the first fragment.
- **Mode three — application on composer:** One fragment is given as argument. In the case of Bind, the call to the Bind composer itself is replaced with the fragment. In the case of Extend, the fragment is added to the collection containing the Extend composer call.

Mode three is only useful in languages that combine the component with the composition language. In such a case, composers are called inside a fragment component "in the place" (the variation point) where a binding or extension should take place. Thus it is often referred to as *in-place* binding (extending).

For the Extend composer, another dimension of variation exists. In unordered collections (e.g., the list of declarations in a Java class definition), it is irrelevant at which

position a fragment is added upon extension. In ordered collections (e.g., the list of statements in a Java method definition), it is important whether a fragment is *prepended* or *appended* to the indicated extension point. In such a case, it has to be specified if the Extend composer works in prepend or append mode.

In some cases, primitive composers may take a list of fragments instead of a single fragment as argument and apply their operation to every element in the list. An Extend, for instance, can extend a hook with all fragments in a list.

As observed, all possible complex composition operations that can be performed on fragment components can be reduced to a set of primitive operations [Aßm03, p. 132]. In Chapter 3, evidence for this statement is provided when we examine complex composers.

2.3.2 Minimal Composition Language

Which composers are called during a composition process is defined in composition programs written in a composition language. To write useful composition programs, the composition language needs to have certain capabilities. There is a minimum of three functionalities a composition language requires to describe executable compositions in an invasive composition system.

1. **Composer calls:** The most fundamental functionality is the ability to call composition operators and pass them the required arguments to execute a composition.

2. **Fragment retrieval:** Before fragments can be composed, they need to be retrieved first. A mechanism has to be provided that allows to retrieve fragments from a given location (e.g., a file on a filesystem) and to address the retrieved fragments in order to pass them as arguments to composers.

3. **Program printing:** To further process a composed program (e.g., by native language tools), the composition language has to provide the functionality of printing back the composition result.

Listing 2.7 describes a minimal composition language. Fragments are loaded and named using the construct `FragmentDeclaration` and printed back by `PrintBack`s. There are different `ComposerCall`s. In fact, every composer has a corresponding call construct in the composition language. The presented minimal composition language supports the universal primitive composers Bind (`Bind`) and Extend (`Extend`).

```
1  CompositionProgram   ::= CompositionStatement+;
2
3  CompositionStatement ::= ComposerCall | FragmentDeclaration | PrintBack;
4
5  FragmentDeclaration  ::= "fragment" FragmentType FragmentName "=" FragmentLocation ";";
6
7  PrintBack            ::= "print" FragmentName "to" FragmentLocation ";";
8
9  ComposerCall         ::= Bind | Extend;
10
11 Bind                 ::= "bind"   SlotName "on" FragmentName "with" FragmentName ";";
12
13 Extend               ::= "extend" HookName "on" FragmentName "with" FragmentName ";";
```

Listing 2.7: Minimal composition language

Using this language, we write a composition program (Listing 2.8) that composes the Java `ClassDeclaration` from Listing 2.4 (saved in the file `/Car.rjava`) with the `Method Declaration` from Listing 2.3 (saved in the file `/DriveMethod.rjava`). The result corresponds to the complete Java program shown in Listing 2.2.

```
1 fragment java.ClassDeclaration   carClass     = /Car.rjava;
2 fragment java.MethodDeclaration  driveMethod  = /DriveMethod.rjava;
3
4 bind mySlot on carClass with driveMethod;
5
6 print carClass to /Car.java;
```

Listing 2.8: Composition program

Figure 2.4 summarizes the properties of an invasive composition system. Based on these properties, we describe an architecture for an implementation of a tool that can generate invasive composition systems and execute invasive compositions. This architecture is presented in the next section.

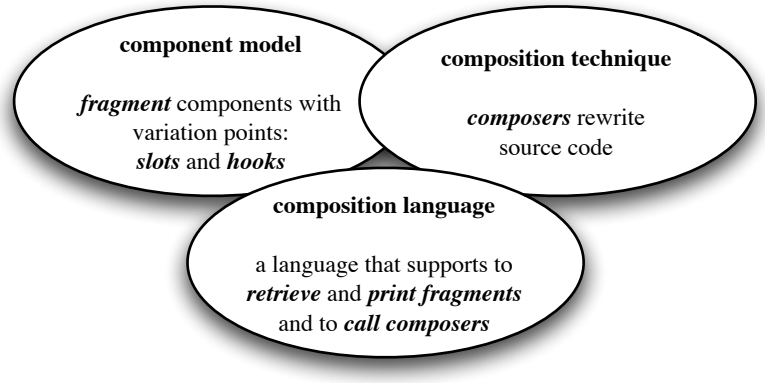

Figure 2.4: Invasive composition system

2.4 Realization

Based on the idea of reuse languages, a language independent tool can be realized that generates invasive composition systems and executes compositions using the primitive composition operators *Bind* and *Extend*. E-CoMoGen [AJH+06] is such a tool. Here, we propose an implementation independent architecture for a composition tool in Section 2.4.1. Then, in Section 2.4.2, we shortly discuss E-CoMoGen as a concrete implementation of this architecture.

2.4.1 Tool Architecture

A composition tool that generates invasive composition systems based on language descriptions and provides the primitive composition operators for any language must have the following capabilities:

1. It must be able to understand language descriptions to automatically integrate support for any given language. In this context, distinction between a core and a reuse language has to be made and the concepts of *slots* and *hooks* occurring in reuse languages have to be understood.

2. It should implement an algorithm to derive a reuse language from a given language or should support the ability to define a reuse language (as an extension to a given language) manually.

3. It has to implement the primitive composition operators to work on fragments written in any given language.

4. It has to understand and provide interpretation for one (or more) composition languages. In particular, it has to interpret the constructs to retrieve components, to print back composition results, and to call composition operators.

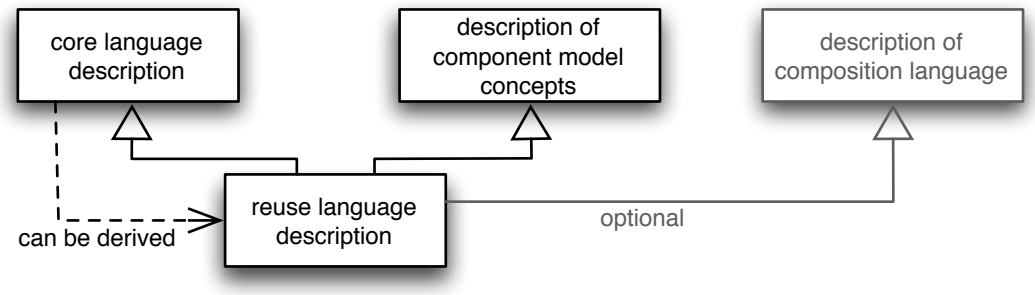

Figure 2.5: Language descriptions in a composition tool

The architecture is divided into two parts. The first part handles the generation of an invasive composition system based on language descriptions. The actual composition is executed by the second part that provides interpretation for the composition language and implements the primitive composers.

The tool architecture is sketched in Figures 2.5 and 2.6. Figure 2.5 shows that a reuse language description has to inherit from a common description of the component model concepts of slots and hooks. The tool can then understand these concepts by knowing about the common description. A reuse language might also inherit from the description of the composition language (Section 2.3) to include in-place Bind and Extend calls. Figure 2.6 illustrates that interpretation for constructs of the composition language and the primitive composers has to be implemented using a native programming language. Composition programs can then be understood and executed by the tool. The next section describes the architecture in more detail by looking at a concrete implementation.

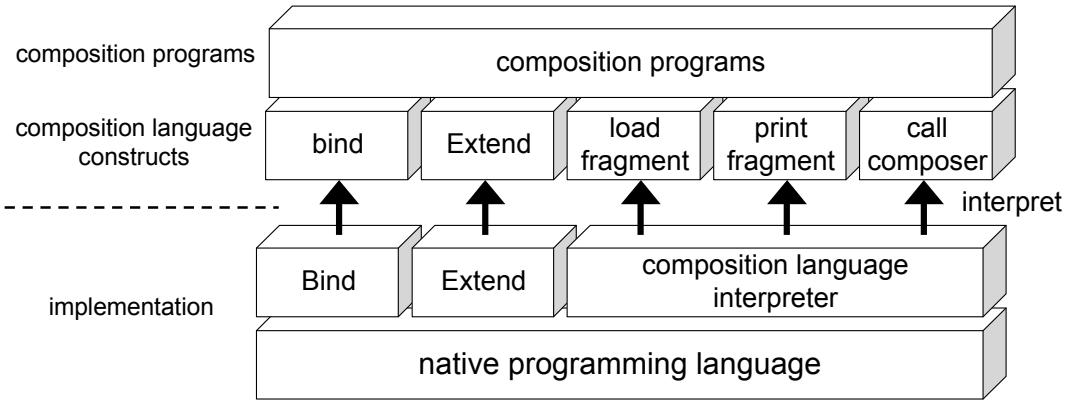

Figure 2.6: Composition tool architecture

2.4.2 Implementation

The implementation we examine here is the tool E-CoMoGen [AJH+06] we described in more detail in [Joh06]. Here, we reconsider the tool's structure with respect to the architecture presented. The different parts of the architecture are described and their implementations are briefly examined to give the reader an overview understanding about the tool.

At first, we look at how languages are described in E-CoMoGen. In general, two possibilities to describe languages are grammars (e.g., using EBNF [Int96]) and metamodels (e.g., using MOF and UML [Gro06a, Gro06b]). While grammars are often easier to use and to understand (when describing languages), metamodels offer more expressiveness[6]. In E-CoMoGen, languages are described using metamodels understood by the

[6]Discussions about language modeling aspects are out of the scope of this thesis. For detailed discussions about language modeling, consult [Joh06] and [Bür05].

Eclipse Modeling Framework (EMF) [BBM03, Fou06]. However, a grammar language to describe languages is also available[7] — grammars are mapped to metamodels internally.

The ability of metamodels defined in EMF to import each other is used to realize the common description of component model concepts (Figure 2.5). A metamodel describing these concepts is defined once (Figure 2.7) and imported into any metamodel describing a reuse language. Figure 2.8 shows a metamodel of a reuse Java (corresponding to the extended Java described by the grammar in Listing 2.5). It is based on the core language description and the component model concepts (compare Figure 2.5).

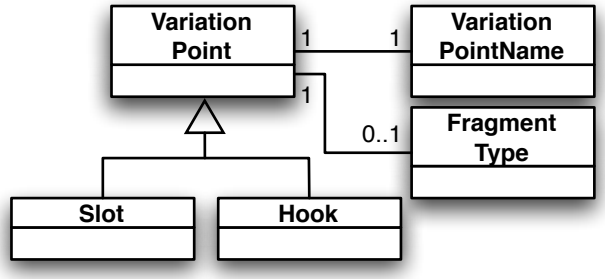

Figure 2.7: Metamodel of common component model concepts

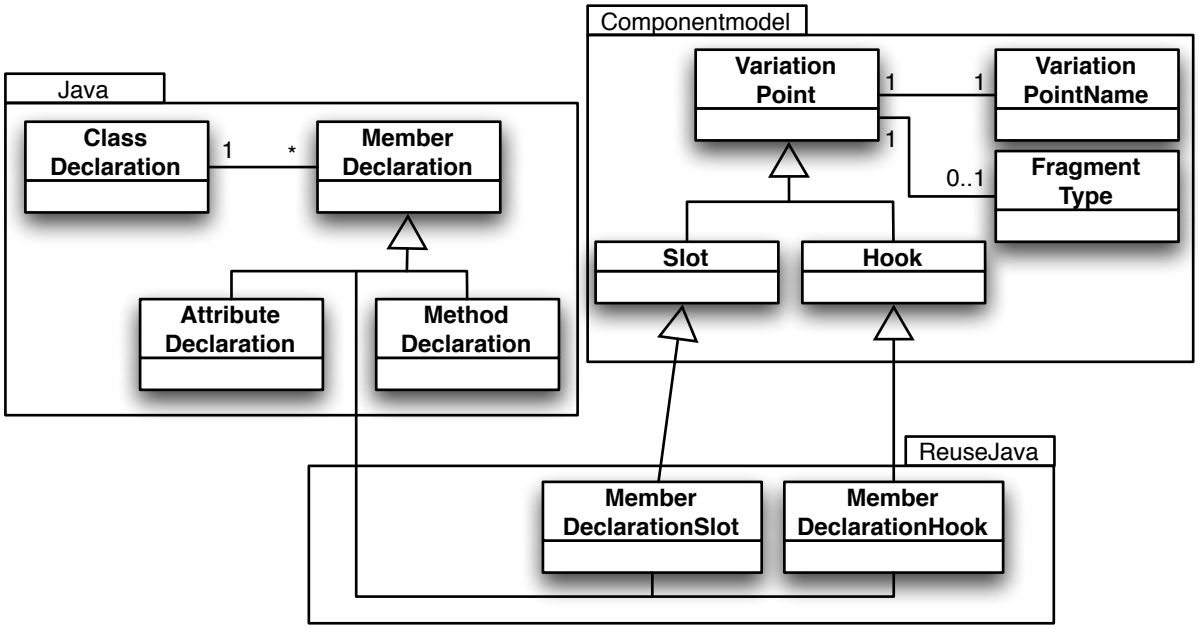

Figure 2.8: Metamodel of a Java based reuse language

[7] In fact, E-CoMoGen also relies on an EBNF like grammar language to describe the concrete syntax of languages (see Appendix B). This is necessary to generate parsers and printers for core and reuse languages.

In our (EMF based) metamodels that describe languages, each metamodel class represents a language construct. An instance of such a metamodel class (i.e., a model element) corresponds to a fragment. An instance of the complete metamodel (i.e., a model) corresponds to a valid program. EMF offers tooling to manipulate models and model elements (now corresponding to fragments), which is used in E-CoMoGen to manipulate fragments as a means to execute compositions.

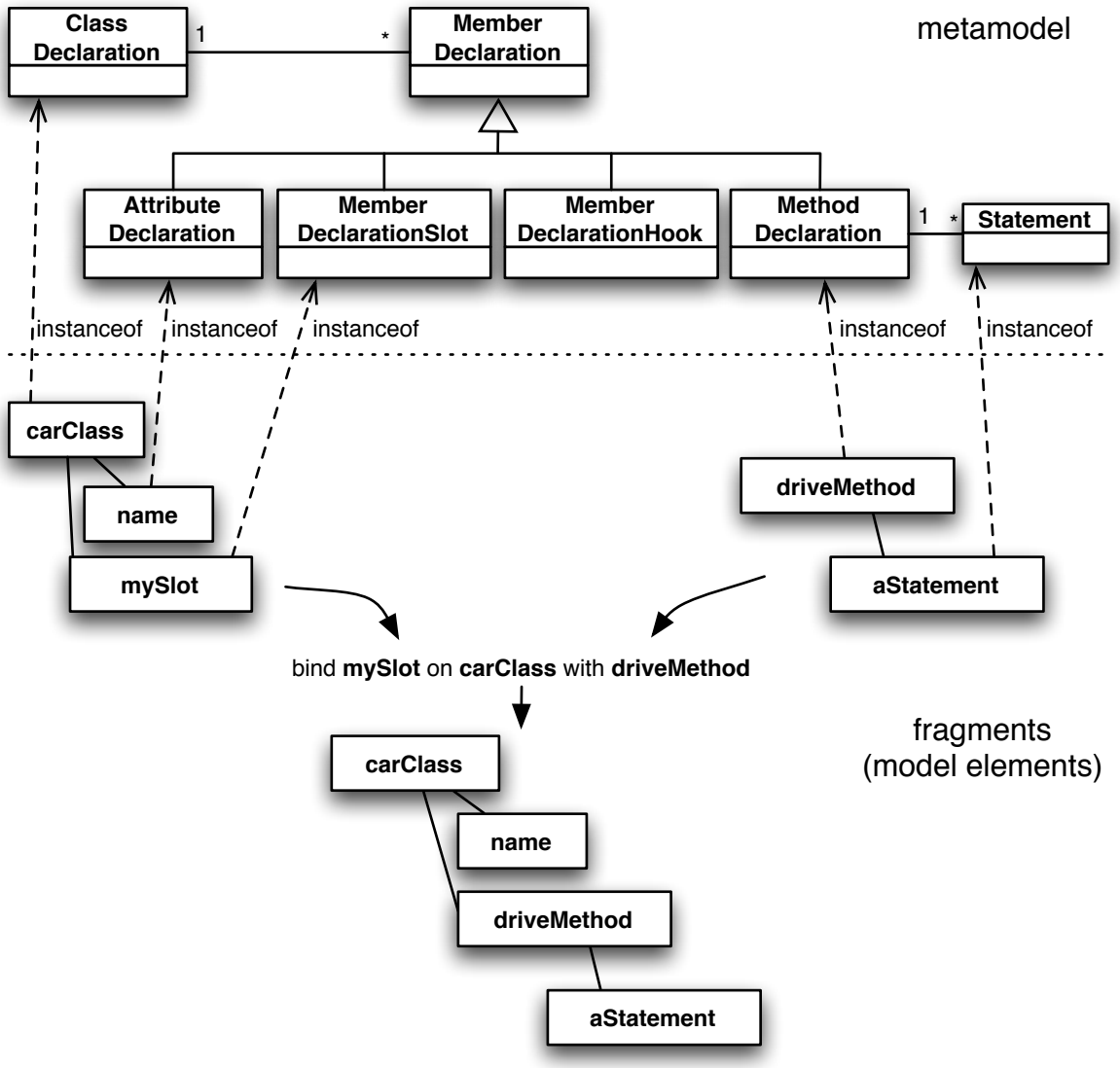

Figure 2.9: Execution of a bind composition

A bind operation, for instance, replaces a slot (i.e., a model element) in a fragment (i.e., a model element) with another fragment (i.e., a model element). The slot can be identified as such because it is the instance of a class in the metamodel that inherits from the common slot class. Figure 2.9 visualizes the process when executing the composition program from Listing 2.8.

The primitive composers and the composition language interpreter (Figure 2.6) are implemented using Java together with the EMF tooling. In order to access the metamodels in Java, code representations of the metamodels are generated by EMF. The bind composer, for example, is implemented in Java as sketched in Listing 2.9.

```
1  public void execute(Composer composer, Map<String, EObject> env,
2                Map<String, EClass> envTypes, List<FileProcessingProblem> problems) {
3
4    Bind bind = (Bind) composer;
5
6    FragmentName fragmentName  = (FragmentName)bind.getFragment();
7    SlotName     slotName      = bind.getSlot();
8    Value        value         = bind.getValue();
9
10   EObject fragmentWithSlots  = resolveName(fragmentName, env, envTypes, problems);
11   List<Slot> slots           = findSlots(slotName, fragmentWithSlots);
12   EObject fragmentToBind     = resolveValue(value, env, envTypes, problems);
13
14   for (EObject slot : slots) {
15     EObject fCopy = (EObject) copyFragment(fragmentToBind);
16
17     EObject container = slot.eContainer();
18
19     Object eo = container.eGet(slot.eContainmentFeature());
20     if (eo instanceof EList) {
21       EList el = (EList) eo;
22       el.add(el.indexOf(slot), fCopy);
23       el.remove(slot);
24     } else {
25       position.eContainer().eSet(slot.eContainmentFeature(), fCopy);
26     }
27   }
28 }
```

Listing 2.9: Bind composer

The object `bind` (Line 4) is an instance of the class `Bind` that represents the corresponding construct of the composition language. Consequently, it contains references to the name of the fragment containing the slot, the name of the slot, and the value — a fragment name or fragment code — to be bound to the slot (Lines 6-8). After resolving fragment names to actual fragments and querying for the slots on the fragments (Lines 10-12), all slots that matched the slot name are bound with a copy (Line 15) of the fragment designated to be bound. For that, the fragment containing the slot is determined (Line 17). Depending on whether the slot is part of a list of fragments, the replacement of the slot with the fragment copy is implemented in different ways (Lines 19-26). Note that all fragments are `EObjects` (EMF objects) because they are instances of an EMF metamodel. The `bind` object is an `EObject` as well since it is a fragment of a composition program.

In the state described in this section, the tool is able to execute simple composition programs written in a composition language that meets the minimal requirements from Section 2.3.2 using the primitive composers. In Chapter 3, we analyze more complex composition approaches and operators. Furthermore, we examine how the tool architecture needs to be extended to support the definition of complex composition operators and richer composition languages.

2.5 Related Work

The idea of invasive software composition was first introduced in [Aßm03]. The book describes the main idea in detail and is thus a good source of information on the topic. Some differences in the terminology exist that we do not want to leave unmentioned. Firstly, the term *slot* is not used at all. Variation points are all called *hooks* alike. Recently, it became clear that it is helpful to distinguish between variation points for genericity and extensibility in the terminology and thus it was decided to use the distinct terms *hook* and *slot*. This terminology was already used in [S$^+$04, AJHS06, Joh06]. Secondly, the terms *primitive composer* and *complex composer* never occurred in the context of invasive composition before. A distinction was rather made between *basic composers* and *compound composers*, where the latter are combinations of the earlier. The set of basic composers contains, in addition to the primitive ones, some composers that implement functionality which is in our view better placed in the composition language.

The idea of a component model based on fragments originates from the object-oriented programming language BETA [LMMPN93]. In BETA, modularization is not integrated into the language itself. Instead, an additional language — *the fragment language* (of BETA) — is provided to define fragments of BETA programs. These fragments can be partial by containing slots. The notion of fragment and slot corresponds to our terminology. The separation between language and fragment language in BETA is a specialization of the separation of core language (in this case BETA) and reuse language (in this case BETA's fragment language).

3 Complex Composition

In the field of software engineering, different composition approaches were studied and realized during the last years. This chapter analyses some of these approaches by looking at their composition systems. Furthermore, the composition systems are converted to invasive composition systems and the contained complex composition operators are identified. The knowledge about invasive composition gained in the previous chapter is employed for these tasks.

On the technical side, we examine how complex composer can be realized. Therefore, the composition tool architecture from the last chapter is enhanced and the tool implementation is extended to meet the requirements of the new architecture.

This chapter starts out by introducing existing composition approaches in Section 3.1 and analyzing their component models in Section 3.2. We look at complex composition operators from the invasive composition viewpoint in Section 3.3 and continue by integrating support for complex composition operators into the composition tool architecture and its implementation in Section 3.4. This chapter again concludes with pointing at related work in Section 3.5.

3.1 Complex Composition Approaches in Software Engineering

In this section we take a brief look at six composition approaches that have achieved a certain amount of popularity. Most of them define a form of gray-box composition. Rather than examining the approaches in full detail, the basic properties that distinguishes one approach from another are pointed out. Also, each approach is interpreted as composition systems.

We start by looking at the mentioned *aspect-oriented programming* and move on to a related approach called *hyperspace programming*. Next, we describe two approaches which have relationships to object-oriented inheritance — *mixin-based programming* and *view-based programming*. *Architectural description languages* are used for black-box composition but can be realized as gray-box compositions as well. Thus, they are also important to us. Finally, we present the ideas of using *design patterns* as composition operators.

3.1.1 Aspect-Oriented Programming

When it comes to identifying components in systems, one speaks about *separation of concerns*. A concern groups a set of related functionality.

Separation of concerns in software systems becomes problematical when the functionality of certain concerns (e.g., security concerns) is scattered all over the system. Such concerns are *cross-cutting* since they touch many elements throughout the whole system.

To separate cross-cutting concerns from the *core* system, the ideas of *aspects* — representing cross-cutting concerns — and *aspect weaving* — integrating an aspect into a core system — were developed [KLM+97]. The approach is called aspect-oriented programming (AOP) and AspectJ [CC03] (see also Section 2.1.2) is the most popular (Java based) implementation available.

The composition technique of AOP is the weaving of aspects into a core and the corresponding composition operator is an *aspect weaver*. The *weaving language* is used to define aspects and to select join points where aspects are woven into the core. It assembles the composition language and a part of the component language (i.e., the part to define aspects). A separate component language to define core components exists as well (e.g., Java in AspectJ).

3.1.2 Hyperspace Programming

Another approach that triggers the problem of cross-cutting concerns is *hyperspace programming* (HP) [OT00b]. Unlike AOP, it is not fixed to the notion of core and aspects but rather defines a *muti-dimensional* separation of concerns. That is, a system can be decomposed in different dimensions depending on where a concern is situated in or how it is distributed throughout the system. A concern's functionality is grouped into a *hyper slice*. Hyper slices are composed into *hyper modules* by defining the *relationships* between the slices to compose.

The reference implementation of HP is Hyper/J [OT00b, OT00a] that implements the hyperspace approach for Java. Like AspectJ, it performs the composition in a preprocessing step but by rewriting binary class files rather than source code[1].

The component model of HP consists of the language to define hyper slices and the default composition interface that exists on a hyper slice. The composition interface is flexible and provides information about a slice's inner structure to enable the definition of different relationships in hyper modules. One could say that each relationship is interpreted by a composer. Then, one composer exists for each relationship in a hyper space composition system. The composition language is the language to define hyper modules.

[1] It is stated in [OT00a] that Hyper/J works non-invasively referring to the fact that the source code is not required to compose classes. However, our understanding of *invasive* in *invasive composition* is the rewriting of components at composition time, independent of the components' format. Thus, the rewriting of binary components (applied in Hyper/J) upon composition is indeed *invasive* in our understanding.

3.1.3 View-Based Programming

If (a part of) a system is used in different contexts, different views on the system exist dependent on the context. *View based programming* (VBP) [Aßm03] follows the idea to define a core that encapsulates the functionality of a system (e.g., a class) and different *views* as extensions to this core.

Different views of a core may coexist as core extensions without interfering with each other. A view does not know about or refer to other views. In contrast to aspects in AOP, views do not encapsulate system functionality. They merely provide alternative viewpoints on the system without influencing its behavior. Another difference is that views always refer to the whole system, while an aspect often only interacts with a part of the system.

A composition system for VBP contains a component model with a component language to define cores and views. The composition interface consists of implicit extension points in the cores. The composition language can be integrated in the language for view definition but might also be separated. The composition technique can be object linking at runtime or can work invasively. In the special case where the core is likely to be changed during system evolution, the composition technique can contain the functionality of generating an adaption layer between a core and its views.

3.1.4 Mixin-Based Programming

Mixin-based programming (MBP) [Bra92, BC90] is a special flavor of object-inheritance. *Mixins* are partial classes that can not be instantiated on their own. They can be mixed into complete classes to extend them with a certain feature.

Compared to traditional inheritance, a mixin definition and its integration are separated. The same mixin can thus be applied to extend different classes and one class can be extended by several mixins.

Similar to AOP and VBP component models, a MBP component model provides a means to define cores (i.e., complete classes) and the smaller mixins. Which mixins are mixed into a class (in which order) is usually specified during class definition. In this case, the composition language is integrated in the language for class (i.e., core) definition. While the concrete composition technique can differ, invasive composition is a good solution here.

3.1.5 Architectural Description Languages

To describe large software systems, it is convenient to have more coarse-grained components that encapsulate sub-systems. An architectural description specifies how these components are connected to a complete system. These descriptions are written in *architectural description languages* (ADLs) [Gar03]. Different ADLs exits for different *architectural styles*.

While the other composition approaches are gray-box based, ADLs typically define black-box compositions. Solely the general architecture of the system is defined without looking into the single sub-system components. The components offer connection points where data flows in or out, which are called *ports*. In-ports are connected to out-ports by *connectors*. Ports can be typed and different connectors with distinct properties may exist (e.g., to transform data from one format into another).

An ADL is a composition language. In a composition system, the corresponding component model specifies how a component may define ports. The composition technique is realized inside connectors. Thus, connectors can be directly understood as composition operators.

3.1.6 Design Patterns

The idea of re-occuring *design patterns* in object-oriented design was first introduced by [GHJV94] and has enjoyed high popularity ever since. Design patterns on their own are not a composition approach or a composition system. However, a pattern defines a structure that can be reused in different places. Thus, it is possible to implement a design pattern as a composition operator [Aßm97].

Such a composition operator can work in an invasive composition system with an object-oriented fragment component model. For instance, the *Observer* pattern [GHJV94] can add the aspect of being observable to a class. This is another solution to separate a cross-cutting concern from the rest of a system: concerns (or aspects) are not defined as a component but inside a composition operator, which differs from the traditional AOP idea.

In [Aßm97] composition operators that implement design patterns to add new aspects to systems are called *aspect composers*. To not confuse them with *weaving composers* of the AOP approach we call them *pattern composers*.

The component model of an invasive composition system with pattern composers is object-oriented. The composition language is most likely a separate script-like language to execute pattern composers.

3.2 Classifications of Component Models

The goal of this section is to provide better understanding of the relationships and similarities between the different composition approaches introduced in the last section and the invasive composition idea. In particular, we look at the different component models and see how they can be related to the fragment based component model of invasive composition. These investigations lead to a better understanding of how complex composers — like the one identified in the last section — are realizable in invasive composition systems. The classification is further used to identify complex composition operators for Semantic Web languages in Chapter 4 and 5.

3.2.1 Component Symmetry

The first property of component models we look at is *component symmetry* [HOT02]. While different aspects of component symmetry are introduced in [HOT02], we concentrate on the meaning of symmetry that refers to the relative size and complexity of components. If all the components that are composed together are of the same dimension in size and complexity the components are symmetric. That is, for instance, the optimal case in hyperspace programming. In aspect-oriented programming we clearly observe component asymmetry — (small) aspects are woven into a (large) core. Figure 3.1 gives an overview of the observed component symmetry in the different approaches

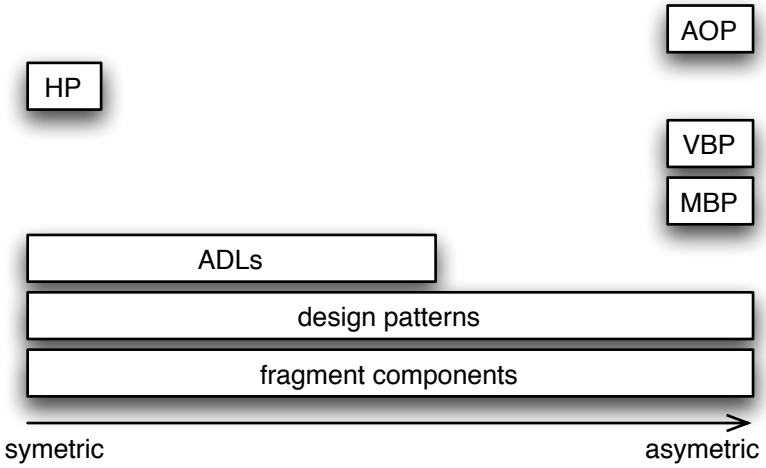

Figure 3.1: Component symmetry of different composition approaches

We note that the fragment component model of invasive composition can be symmetrical but also highly asymmetric if desired. The symmetry depends on the type of fragments that are permitted to exist by the component model. The fragment component models described in Section 2.2 admit all different kinds of fragments a language defines to appear as components. However, such a component model can also be tailored to only permit certain types of fragments.

We conclude from this observations that the different component symmetries, existent in the different composition approaches, can be realized in the component models of invasive composition systems.

> **Observation:** Concerning the relative size and complexity of components, any component symmetry may exist in an invasive composition system.

3.2.2 Application of Genericity and Extensibility

Section 2.2 explained how genericity and extensibility are supported in the fragment component models of invasive composition systems. In general, every component model needs to provide support for genericity or extensibility. Components that can neither be configured nor extended do not offer any composition interface and can thus not be composed.

In this section we analyze the component models of the different composition approaches with respect to genericity and extensibility. Here, we take the position that the component models, while keeping their original semantics, are designed as if they belong to an invasive software composition system. Being able to describe the genericity and extensibility capabilities of the component models this way is further proof that they can exist in an invasive composition system.

- **AOP:** Aspects are woven into a core at different join points. In this process, information is added to the core — it is extended — and the original functionality is not changed. Aspects themselves, however, can be parameterized with information gathered from the core at one specific join point (e.g., the name of the method the aspect is woven into). Genericity is applied to keep certain points in an aspect parameterizable. AOP relies on extensibility (for aspect weaving) and genericity (to parameterize aspects).

- **HP:** A hyper slice always contains a whole concern. The grouping of hyper slices into a hyper module can be understood as extending hyper slices with hyper slices one after another. Hype slices, which are the only components described by the component model, do not include generic constructs. Thus, HP relies solely on extensibility.

- **VBP:** A view is defined for a core and extends the core. Unlike aspects, views are not necessarily configurable. VBP relies mainly on extensibility.

- **MBP:** Like a view, a mixin extends a core without varying the original core or the mixin itself. MBP relies on extensibility only.

- **ADLs:** Connectors connect defined ports. When connectors work invasively, ports disappear upon connection (i.e., composition) and the connected components are merged by parameterization and extension. Thus, ADLs require primarily genericity but also use extensibility.

- **design patterns:** Different design patterns have distinct properties. Hence, components in a composition system with pattern composers require a very flexible composition interface. The design pattern composition approach requires genericity and extensibility.

Observation: All extensibility and genericity capabilities required by the component models of the described composition approaches exist in invasive composition systems.

3.2.3 Composition Interface

From the observations made in the last section follows that components of any approach contain, from the invasive viewpoint, a composition interface that contains either slots, declared hooks, or implicit hooks. The different approaches require these variation points addressed in distinct ways. Some allow for a flexible selection of variation points for composition (e.g., defining a slot to bind), others include composers that identify the set of variation points they work on themselves (e.g., mixing a mixin into a class). Figure 3.2 illustrates the requirements of the different composition approaches.

	primitive Bind	primitive Extend	AOP	HP	VBP	MBP	ADLs	d.p.
slot binding								
selectable	x	-	x	-	-	-	x	x
composer dependent	-	-	-	-	-	-	-	x
declared hook extension								
selectable	-	x	-	-	-	-	x	x
composer dependent	-	-	-	-	-	-	-	x
implicit hook extension								
selectable	-	-	x	-	-	-	-	x
composer dependent	-	-	x	x	x	x	x	x

Figure 3.2: Composition interfaces requirements of composition approaches

Approaches that rely on extensibility only (HP, VBO, MBP) do not use slots or declared hooks. Usually, no variation points are directly addressed in composition programs. They are determined by the composers (e.g., when a mixin is mixed into a class, the composer knows where to extend the class with the mixin). In AOP, hooks may be addressed directly as weaving points (see Section 3.3.2). However, the weaver of AspectJ determines the extension points itself (see Section 3.3.3). Additionally, AOP can make use of slots and declared hooks in aspects. ADLs require a declared component interface (slots and declared hooks) as well to enable the declaration of ports, which are addressed directly in architectural descriptions. A design pattern composition system should have access to the full flexibility of a fragment based component system.

Observation: Slots and declared hooks can act as other kinds of variation points (e.g., ports). Implicit hooks can describe any default composition interface that is apt for extension (e.g., possible weaving points).

3.3 Complex Composition Operators

We demonstrated how component models of different composition approaches can be reinterpreted as component models of invasive composition systems. The next step is to realize the complex composition operators and to define appropriate composition languages for these systems. Then, any composition system identified in the different composition approaches can be realized completely as an invasive composition system using an invasive composition tool. Elements of these invasive composition system, in particular complex composers, can then be transferred to new invasive composition systems, which work, for instance, with ontology fragment components.

A method to describe complex composers is defined in Section 3.3.1. The method is used to define complex composers for aspect-oriented invasive composition systems in Sections 3.3.2 and 3.3.3.

3.3.1 A Language to Define Composition Operators

A complex composer can be defined as a sequence of compositions performed by primitive composers [Aßm03, p. 132]. It is obvious that a language is needed to define such a sequence. Since we already support different languages in the tool architecture from Section 2.4.1, we examine those languages to find an appropriate language among them for composer definitions.

The existing architecture already contains a language used to define primitive composers: the native language in which the composition tool is implemented (Java in the case of E-CoMoGen). An obvious conclusion is to follow this idea and implement additional composers in the native tool language as well. In this case, the architecture can almost stay like it is — we merely need extensibility on the native level to add new composers. The calls to other (primitive) composers can then take place on the native level (e.g., by Java method calls).

The benefits of using a general purpose language for composer definition are the flexibility, richness, and support for reuse in such a language. One can on the one hand reuse other composers — in particular the primitive ones — and on the other hand reuse other functionality provided in libraries (e.g., Java class libraries).

The problem with this approach is that the more flexible and general a language is, the more error-prone the programs become. The language gives less guidance — and not enough restrictions — to the programmer, which is a premise to ugly and erroneous code.

We believe that composer developers should have a language at hand that is tailored for the purpose of defining complex composers. Such a language guides developers by only providing them with constructs designed for the purpose of defining composers. The composition tool itself is able to process and manage such composer definitions.

Hence, the developer does not need any knowledge about the tool's internals and the native language and frameworks it applies (e.g., Java and EMF in E-CoMoGen).

The language described above already exists in our composition tool — the *composition language*. It is apt to write compositions using the primitive composers and other convenient constructs required to define meaningful compositions. We know compositions defined in the composition language as *composition programs*.

Complex composer definitions are very similar to composition programs, despite the following differences:

1. **Metalevel:** Composition programs are programs consisting of a set of fragments that are instances of constructs of a composition language. Composition operators are constructs of a composition language. When used, they are instantiated to fragments (Figure 3.4).

2. **Reusability:** A composition program is not necessarily reusable. A composition operator is always reused when it is instantiated.

3. **Configurability:** A composition program is not configurable in the first place. Composition operators have to be configurable to be usable in different contexts.

Regarding the second two points one may argue that composition programs may contain functions or procedures to reuse configurable parts. While possible, those are not necessary requirements to be supported by a composition language (Section 2.3.2).

If we extend our composition language to be applicable for composition operator definitions as well (Figure 3.3), the metalevel difference is the most crucial point to address. A composition tool architecture has to provide a feature that establishes the connection (Figure 3.4) between a composer definition (a composition program) and the composer construct (a composition language construct).

New composers require a composition language with constructs to call them. Such a composition language can reuse elements of the existing one (i.e., extend it). Figure 3.3 illustrates the different extensions of the existing composition language. The next section explains these relationships in more detail on a concrete example.

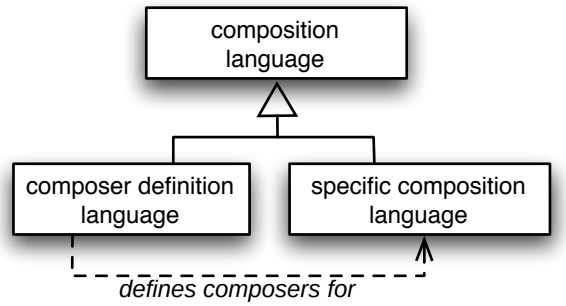

Figure 3.3: Composition language extensions

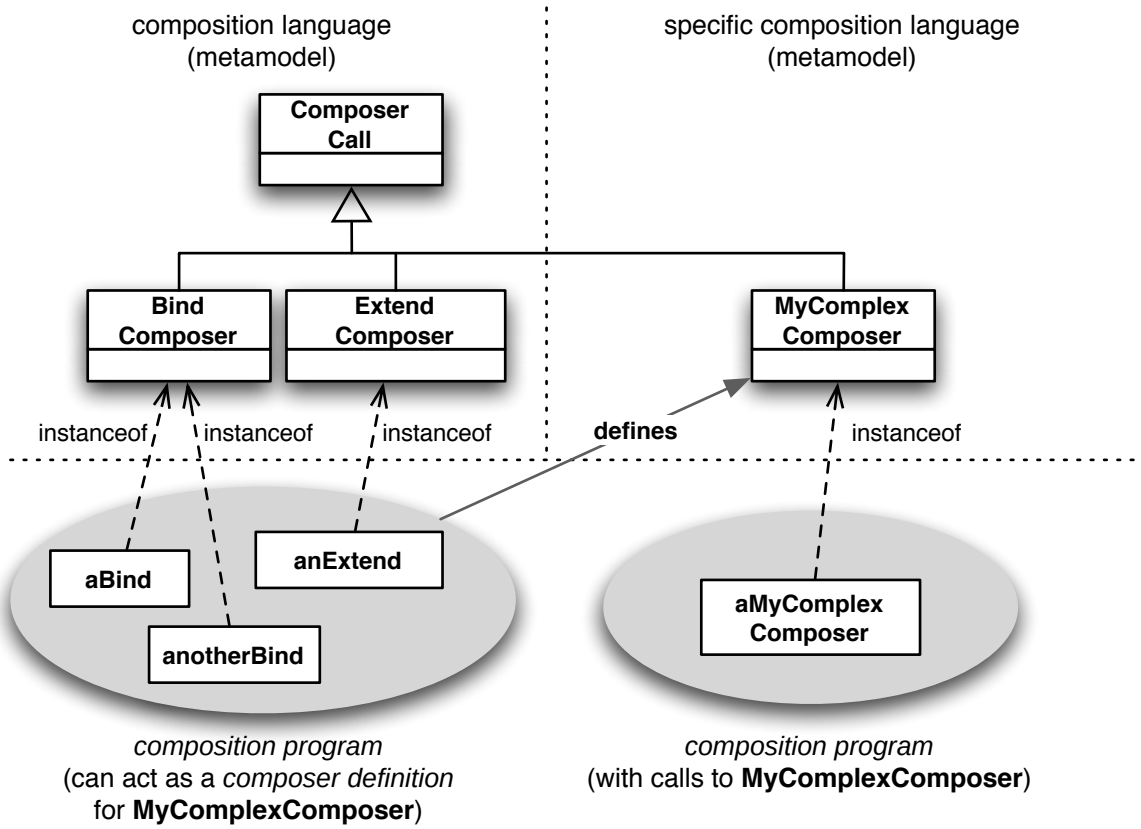

Figure 3.4: Composition programs and composer definitions

3.3.2 The Complex Composition Operator Weave

We are now prepared to define a complex composition operator for aspect weaving. Listing 3.1 contains the definition. The language used is an extension of the composition language from Section 2.3.2 that is enriched with some convenient functionality. It is the *basic composition language* of E-CoMoGen defined in Appendix A.

```
1  define composer weavinglanguage.Weave(type, core, position, qualifier, aspect) {
2      foreach (thisPoint : ->core) {
3          if (thisPoint instanceof ->type) {
4              if (qualifier instanceof weavinglanguage.Before) {
5                  prepend thisPoint.->position with ->aspect;
6              }
7              if (qualifier instanceof weavinglanguage.After) {
8                  append thisPoint.->position with ->aspect;
9              }
10         }
11     }
12 }
```

Listing 3.1: Complex composition operator Weave

The weaver defined above is language independent. The definition only references types (i.e., constructs) of the weaving language (`weavinglanguage.*`) and no types of other languages. Language dependent type information is dynamically passed to the composer during execution in its arguments.

Line 1 states that the composer that is defined is executed using the `Weave` construct of the language `weavinglanguage` (Figure 3.5). The arguments (`type`, `core`, `position`, `qualifier`, and `aspect`) correspond to the references of `Weave` (Figure 3.5). The values passed to a composer are fragments (i.e., instances of the corresponding weaving language constructs).

The weaving composer combines pointcut and aspect definitions. Pointcut definition is limited to defining the type of fragments that are cross-cutted (`type` argument) and the core containing these fragments (`core` argument). Additionally, the exact point of weaving has to be specified by the `position` (a position in a list of fragments) and the `qualifier` arguments.

The weaver iterates the core and looks for join points (i.e., fragments) of the indicated `type` (Line 3). If one is found, the `aspect` is bound to the specified position in the join point (Lines 4–9). Note that the join point itself is made available as `thisPoint` (Line 2) and can be accessed by the aspect to retrieve core dependent information, as done in Listing 3.3.

For the composer we define a composition language — the weaving language — in Figure 3.5. The language reuses parts of the basic composition language but adds one construct to call the weaving composer (`Weave`) and one to write (weaving) composition programs (`WeavingProgram`).

Since the weaver is language independent, it can be used without modification in any invasive composition system. In the following example, we apply it in a Java based invasive composition system.

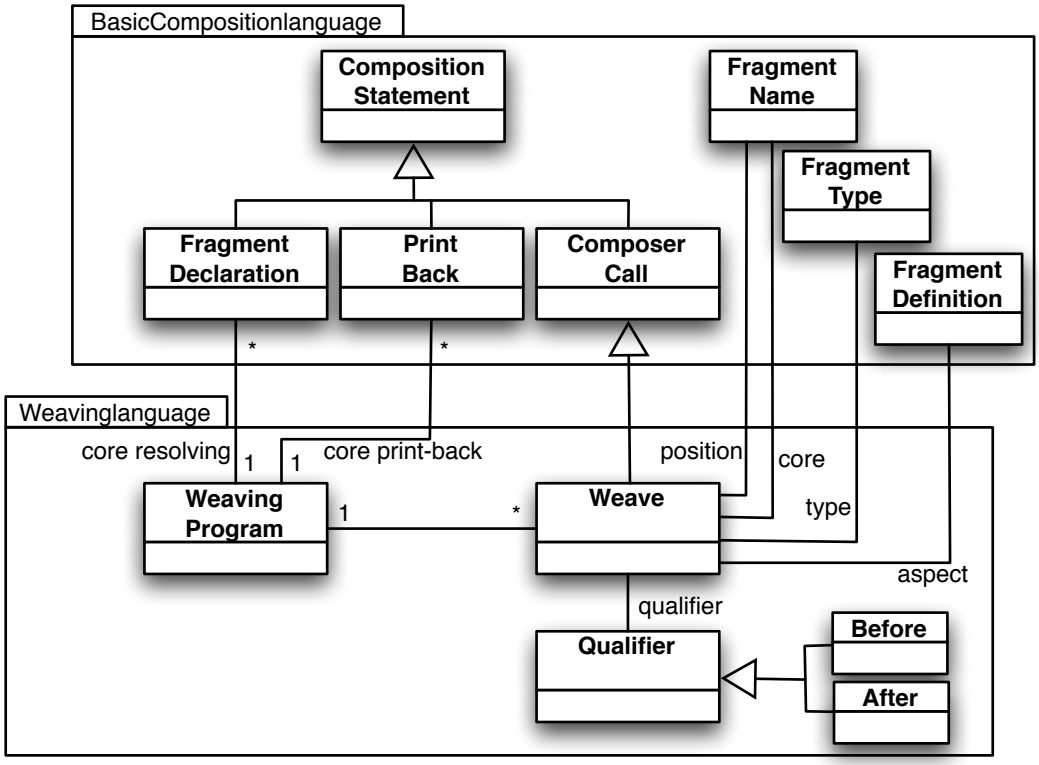

Figure 3.5: Weaving language as refinement of the composition language

Listing 3.2 shows a Java[2] compilation unit containing one class declaration that in turn contains several attribute and method declarations. A weaving defined in Listing 3.3 weaves a debugging aspect into all `MethodDeclarations` in the `classBody` of the first (and only) `classDeclaration`. The result of the operation is listed in Listing 3.4

```
1  package org.foo.vehicles;
2
3  import org.foo.FuelStation;
4  import org.foo.Engine;
5
6  public class Car extends Vehicle {
7          private String name;
8
9          private Engine engine;
10
11         public void drive(int direction) {
12                 turn(direction);
13                 accelerate();
14         }
15
16         public void startEngine() {
17                 engine.start();
18         }
19
20         public void stopEngine() {
21                 engine.stop();
22         }
23
24         public void takeFuel(int liters) {
25                 FuelStation.get().fuel(liters);
26         }
27 }
```

Listing 3.2: Java class

```
1  cores:
2
3    core java.CompilationUnit myClass = /Car.java;
4
5  weavings:
6
7    weave java.MethodDeclaration in myClass.classDeclarations[0].classBody
8                                                  before statements[0] {
9        'System.out.println("Entering Method: ' +
10         thisPoint.name + '(' + thisPoint.arguments + ')");'.java
11   }
12
13   weave java.MethodDeclaration in myClass.classDeclarations[0].classBody
14                                                  after statements[-1] {
15       'System.out.println("Leaving Method:  ' +
16         thisPoint.name + '(' + thisPoint.arguments + ')");'.java
17   }
18
19 output:
20
21   print myClass to /Car.java;
```

Listing 3.3: Weaving of a debug aspect into a Java class using Weave

[2] In fact, the language is a subset of Java: *Java-* (Appendix B.7).

```
1  package org.foo.vehicles;
2
3  import org.foo.FuelStation;
4  import org.foo.Engine;
5
6  public class Car extends Vehicle {
7          private String name;
8
9          private Engine engine;
10
11         public void drive(int direction) {
12                 System.out.println("Entering Method: drive(int direction)");
13                 turn(direction);
14                 accelerate();
15                 System.out.println("Leaving Method:  drive(int direction)");
16         }
17
18         public void startEngine() {
19                 System.out.println("Entering Method: startEngine()");
20                 engine.start();
21                 System.out.println("Leaving Method:  startEngine()");
22         }
23
24         public void stopEngine() {
25                 System.out.println("Entering Method: stopEngine()");
26                 engine.stop();
27                 System.out.println("Leaving Method:  stopEngine()");
28         }
29
30         public void takeFuel(int liters) {
31                 System.out.println("Entering Method: takeFuel(int liters)");
32                 FuelStation.get().fuel(liters);
33                 System.out.println("Leaving Method:  takeFuel(int liters)");
34         }
35 }
```

Listing 3.4: Java class with debug aspect

3.3.3 The Complex Composition Operator JavaMethodWeave

The language independent weaver requires the specification of the exact weaving position inside a join point (`position` argument). In contrast, in pointcuts of AspectJ the specification of the join points alone — e.g., a set of of Java methods — is sufficient. The composer knows that weaving before a method execution means to weave the aspect before the first element of the method body. Our language independent weaver can not know about such language dependent details.

However, a Java specific weaver that knows where to weave in Java methods can be defined (Listing 3.5). When we apply this composer in Java weavings we save the specification of exact weaving points inside join points (Listing 3.6).

```
1  define composer weavinglanguage.JavaMethodWeave(core, qualifier, aspect) {
2      foreach (thisPoint : ->core) {
3          if (thisPoint instanceof java.MethodDeclaration) {
4              if (qualifier instanceof weavinglanguage.Before) {
5                  prepend thisPoint.statements[0] with ->aspect;
6              }
7              if (qualifier instanceof weavinglanguage.After) {
8                  append thisPoint.statements[-1] with ->aspect;
9              }
10         }
11     }
12 }
```

Listing 3.5: Complex composition operator JavaMethodWeave

The drawback is that the composer has to know about the Java specific construct `MethodDeclaration` (Line 3) and its structure (e.g., the `statements` reference: Lines 5 and 8). Hence, it is no longer applicable for fragments written in other languages than Java.

```
1  cores:
2
3    core java.CompilationUnit myClass = /Car.java;
4
5  weavings:
6
7    weave before in myClass.classDeclarations[0].classBody {
8        'System.out.println("Entering Method: ' +
9          thisPoint.name + '(' + thisPoint.arguments + ')");'.java
10   }
11
12   weave after  in myClass.classDeclarations[0].classBody {
13       'System.out.println("Leaving Method:  ' +
14         thisPoint.name + '(' + thisPoint.arguments + ')");'.java
15   }
16
17 output:
18
19   print myClass to /Car.java;
```

Listing 3.6: Weaving of a debug aspect into a Java class using JavaMethodWeave

The two different weaving composers demonstrated that language dependent complex composers are required to meet specific composition needs of a certain language. Language independence is only possible as long as a composition technique is abstract enough to be applicable on fragments written in any language. In Chapters 4 and 5 language dependent composers are examined that can not be realized language independently.

3.4 Realization

To actually execute complex composers like the ones defined we extend the composition tool architecture introduced in Section 2.4.1 and its implementation. This is necessary to provide two new functionalities: defining complex composers using a composition language and defining extended composition languages.

3.4.1 Enhanced Tool Architecture

Here, we present a composition tool architecture that supports utilization of an (extended) composition language to define new composers and that defines a modular composition language interpreter. The latter is required to add a new feature to a composition language not yet supported by the composition interpreter. Such features are implemented in the tool's native language. Note that this could also be used to implement new composition operators natively. The architecture is shown in Figure 3.6.

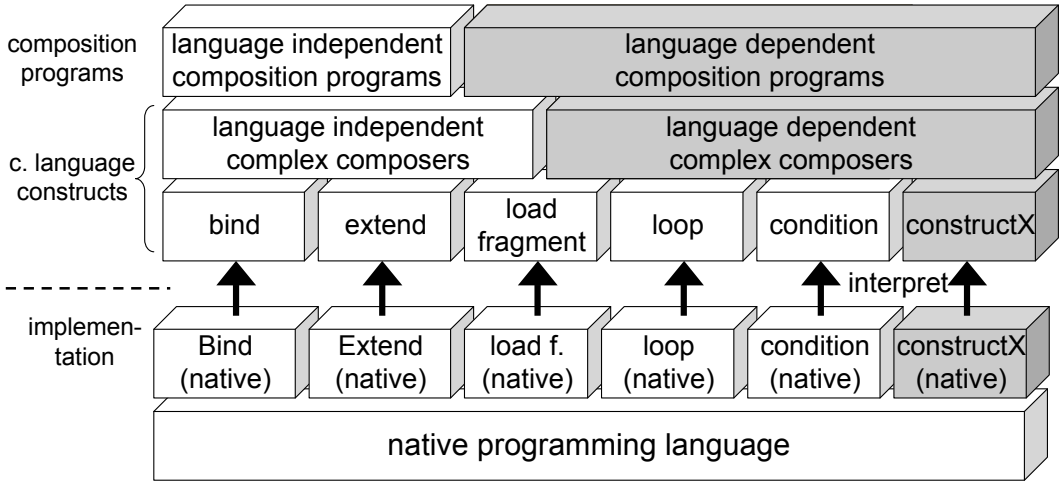

Figure 3.6: Extended composition tool architecture

If a new composer construct is contained in a composition language, the composer has to be defined using the composer definition language (which is an extension of the composition language: Figure 3.3). The tool implementation has to connect the construct with the corresponding definition when a composition is executed.

3.4.2 Implementation

The implementation of E-CoMoGen is modified to support modular interpreter definition. Therefore, the interface `ICompositionLanguageInterpreter` (Listing 3.7) is introduced. Interpreter modules, as well as primitive composers, implement this interface and register the construct they are able to interpret in a registry.

```
1 public interface ICompositionLanguageInterpreter {
2
3     public void interpret(EObject fragment, Map<String, List<EObject>> env,
4             Map<String, EClass> envTypes, List<FileProcessingProblem> problems);
5 }
```

Listing 3.7: Interface for a composition language interpreter module

Implementing the interface means implementing the `interpret` method. An interpreter has access to the current environment of loaded fragments[3]. It can modify the environment or, in case of a primitive composer, modify fragments themselves.

A special interpreter module is the *complex composer interpreter* that realizes the connection between composer constructs and composer definitions. It is called when a non-primitive composer is encountered. The interpreter constructs an environment containing the fragments passed as arguments, looks up the definition of the composer, and feeds the definition together with the environment into the interpreter mechanism (Figure 3.7).

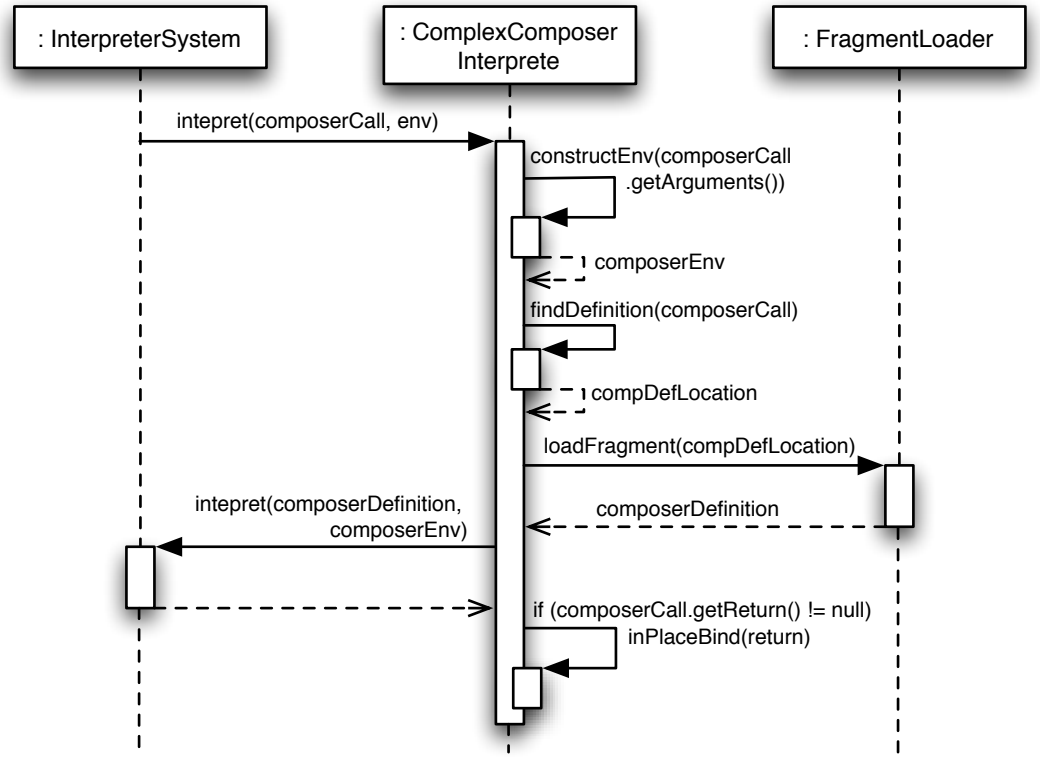

Figure 3.7: Complex composer interpretation

[3]In fact, the environment contains lists of fragments (which might contain only one element). This enables the specification of a set of fragments (e.g., several Java methods) in one file.

New in-place composers can also be defined. They return a fragment as result of the composition. The returned fragment is then bound in-place (Figure 3.7).

The implementation now supports definition and execution of complex composers. The weaving composers and the example from Sections 3.3.2 and 3.3.3 are executable by the tool. The next two chapters use the defined language independent weaver in other invasive composition systems and define new language dependent complex composers. To achieve the results presented in those chapters, the extended tooling has been applied in different experimentations.

3.5 Related Work

The COMPOST system [Con06b] implements the invasive composition principles as a Java framework. The framework supports Java and XML as core languages and is extensible for new languages by means of implementing framework extensions in Java. COMPOST already contains several Java dependent composers based on the composition approaches analyzed in this chapter. However, the framework lacks the flexibility on the language independent level that is provided by E-CoMoGen.

In [Kar06] an aspect-oriented composition system based on COMPOST is introduced. While related to COMPOST's Java composition system the work aims at keeping the aspect-oriented framework extensions reusable for future COMPOST composition systems. Ideas from this work should be taken into account for further development of the (language independent) aspect weaving composer and the weaving language.

Several design patterns were implement as COMPOST composers in [Wüs05]. The work demonstrates how patterns can be realized as complex composers. In Chapter 4 we also implement patterns as composers.

4 Composition of Ontologies

The last chapter talked about complex composition and introduced a tool architecture that supports the definition of complex composition operators. An aspect weaver was introduced as a language independent complex composer. It was demonstrated how this composer could be applied to fragments written in the programming language Java.

This chapter concentrates on complex compositions of fragments written in the ontology language OWL. To construct OWL ontologies, we use the concrete OWL syntax *Notation 3* (N3), which is briefly described in Section 4.1.

We discuss how the results from the last chapter can be utilized for complex ontology composition in Section 4.2. As a first example of compositions of this kind, the aspect weaver from Section 3.3.2 is applied to OWL fragments in Section 4.3. Next, an OWL specific complex composer is defined in Section 4.4. This composer follows the idea of *pattern composers* (Section 3.1.6) but is based on *ontology patterns* [SWM06] rather than design patterns.

4.1 OWL Notation 3: A Short Introduction

The Web Ontology Language (OWL) [SWM04], build on top of the Resource Description Framework (RDF) [KC04], has several syntaxes. While the OWL RDF/XML syntax [BM04] is designed for publishing and exchanging ontologies, another human readable notation — called *Notation 3* (N3) — is available [BL06]. This section gives a short introduction to the parts of N3 that are used in the following sections.

N3 is a concrete syntax for RDF descriptions and OWL ontologies, which are at their base a collection of triples — each consisting of a *subject*, a *predicate* and an *object*. An example of such a triple is `Pat knows Jo .` where `Pat` is the subject, `knows` is the predicate, and `Jo` is the object.

In N3, every predicate, subject, or object is uniquely described by an unified resource identifier (URI) — e.g., `<http://foo.org/ont#Pat>`. The empty URI (`<>`) references the current document. The object can also be a simple string or integer — e.g., `<#Pat> <#hasName> "Pat" .` — where `<#Pat>` and `<#hasName>` are defined in the current document while the object is simply a string (`"Pat"`) . Additionally, N3 offers shorthands for common predicates like **a**, which is an abbreviation for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

Prefixes can be defined as abbreviations for URIs — e.g., `@prefix owl: <http://www.w3.org/2002/07/owl#> .`. The empty prefix is usually defined for the empty URI — `@prefix : <> .`. The example can then be written as `:Pat :knows :Jo .`.

If more than one statement should be made about one subject several predicate-object pairs can be defined separated by `;`. Several objects for the same subject and predicate can be defined separated by `,`.

As example, Listing 4.1 uses N3 to define the classes `Book` and `ForChildren`.

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix :        <> .

:Book
      a               owl:Class ;
      rdfs:subClassOf owl:Thing .

:ForChildren
      a               owl:Class ;
      rdfs:subClassOf owl:Thing .
```

Listing 4.1: Ontology with classes `Book` and `ForChildern`

To learn more about the syntax and capabilities of N3 consult [BL06] and [BLHC05].

4.2 Learning from Existing Ideas

In the previous chapter, different composition approaches from software engineering were studied with the goal to transfer composition ideas to the Semantic Web domain. Concerning ontologies, we first expected direct reusability of many approaches due to similarities between object-oriented and ontology languages (e.g., the concepts of classes and instances). However, it became clear at an early stage of this work that the transfer of approaches between the two domains is not that straight forward.

Programming languages are used to define functionality of an executable system, while ontology languages are applied to describe semantic relations of data. The composition approaches from the last chapter mostly concern the separation of functional concerns in systems. Thus, many practical examples describe functional concerns (e.g., ideas for an aspect library in AspectJ [asp05]). In most cases, similar requirements for concerns in ontologies are hard to identify.

Many of the investigated composition approaches apply and add extensibility capabilities to the underlying component language. Ontology languages, and OWL in particular, do already support a high degree of extensibility. Thus, many benefits of transferring extension based approaches (AOP, HP, VBP, MBP) to the ontology domain are yet to be discovered. Further work in this area needs to be done.

However, there is no support for genericity in OWL and there is no way to describe ontology architectures. Therefore, we see the main benefits in adopting composition approaches that trigger structuring and reuse of structure, namely ADLs and pattern composers, for the ontology domain.

Even if the benefits of aspect weaving in ontologies are not clarified yet, we deploy the language independent weaving composer (Section 3.3.2) for ontology weaving in Section 4.3. This stresses the language independency of the composer and shows that weaving of ontologies is technically possible. Section 4.4 then transfers the pattern composer idea (Section 3.1.6) to the ontology domain and implements an OWL dependent composer that realizes an *ontology pattern* [SWM06].

4.3 Weaving Ontologies

To apply weaving to ontologies, we try to identify (non-functional) cross-cutting ontology aspects. In contrast to object-oriented programing languages, it seems to be easier to isolate a cross-cutting concern in an ontology.

Consider, for instance, the ontology that defines the concepts of book and "thing for children" from Listing 4.1. An ontology that defines some individuals that belong to the class `Book` is shown in Listing 4.2. Being a book is a cross-cutting concern through all individuals in the ontology.

```
:PeterPan
      a             :Book ;
      rdfs:seeAlso  <http://isbn.nu/1419151576> ;
      :bookTitle    "The Adventures Of Peter Pan" .

:TomSawyer
      a             :Book ;
      rdfs:seeAlso  <http://isbn.nu/0195114051> ;
      :bookTitle    "The Adventures Of Tom Sawyer" .

:PippiLongstocking
      a             :Book ;
      rdfs:seeAlso  <http://isbn.nu/0670876127> ;
      :bookTitle    "The Adventures of Pippi Longstocking" .
```

Listing 4.2: Book individuals (in file *books.n3*)

Additionally, we would like to express the fact that all the books are also things for children. If this aspect, which again crosscuts all individuals, should be isolated, we can simply define it separately (Listing 4.3). An ontology reasoner is able to connect the physically separated definitions and draw the correct conclusion that the individuals are both: books and things for children.

```
:PeterPan
      a       :ForChildren .

:TomSawyer
      a       :ForChildren .

:PippiLongstocking
      a       :ForChildren .
```

Listing 4.3: Extension of book individuals

Alternatively, the weaving composer can be used to weave the aspect of "being for children" into the original individual definition. The weaving specification and the composition result are shown in Listings 4.4 and 4.5 respectively.

```
cores:

    core n3.N3Doc books = /books.n3;

weavings:

    weave n3.Triple in books.statements before predicateObjectList[0] {
        'a    :ForChildren ;'.n3
    }

output:

    print books to /booksForChildren.n3;
```

Listing 4.4: Weaving of `ForChildren` aspect

```
:PeterPan
    a             :ForChildren ;
    a             :Book ;
    rdfs:seeAlso  <http://isbn.nu/1419151576> ;
    :bookTitle    "The Adventures Of Peter Pan" .

:TomSawyer
    a             :ForChildren ;
    a             :Book ;
    rdfs:seeAlso  <http://isbn.nu/0195114051> ;
    :bookTitle    "The Adventures Of Tom Sawyer" .

:PippiLongstocking
    a             :ForChildren ;
    a             :Book ;
    rdfs:seeAlso  <http://isbn.nu/0670876127> ;
    :bookTitle    "The Adventures of Pippi Longstocking" .
```

Listing 4.5: Waving result

An advantage of using the weaving composer over the natural way of extending the semantic of individuals is that one fact shared by a large set of individuals can be varied very flexibly. Certainly, a weaving composer for ontologies can look simpler, since the weaving can be achieved by adding new triples without changing the core definition internally. However, the language independent weaver already exists and can thus be reused. Applying it also yields a better human understandable result since all facts about an individual are gathered in one place.

Situations might exist where cross-cutting concerns become more complex and ontology design profits more from the application of weaving composers. Possibly, OWL dependent weavers are necessary in certain cases. More research has to be done in this direction.

This section demonstrated how the weaving composer, which was applied for weaving in Java (Section 3.3.2), can be reused in a language of another domain. In the next section we design an OWL specific composition operator.

4.4 OWL Specific Complex Composition Operator: Ontology Pattern Composer

In Section 3.1.6 the idea to use design patterns as composition operators was presented. Traditional design patterns are usually applied in the design of object-oriented systems. Nevertheless, as patterns can be found literally everywhere, there are also patterns in ontologies. In [SWM06], common and reusable ontology patterns are documented.

Ontology patterns are meant to give reusable solutions to complex design problems in ontologies. Often, there is more than one solution to the same problem with different advantages and disadvantages. Thus, different patterns present exchangeable solutions to the same problem.

One set of patterns triggers the question of how to represent classes as property values [NUW05]. In the following, we describe this problem and two patterns that target it. These patterns are implemented as complex composers and applied on an ontology.

4.4.1 Classes-as-Property-Values Pattern

In ontologies, it is sometimes convenient to fill a property value with a class (Figure 4.1) in contrast to filling it with an individual. While OWL Full[1] supports this functionality directly, it is not expressible in OWL DL or OWL Lite.

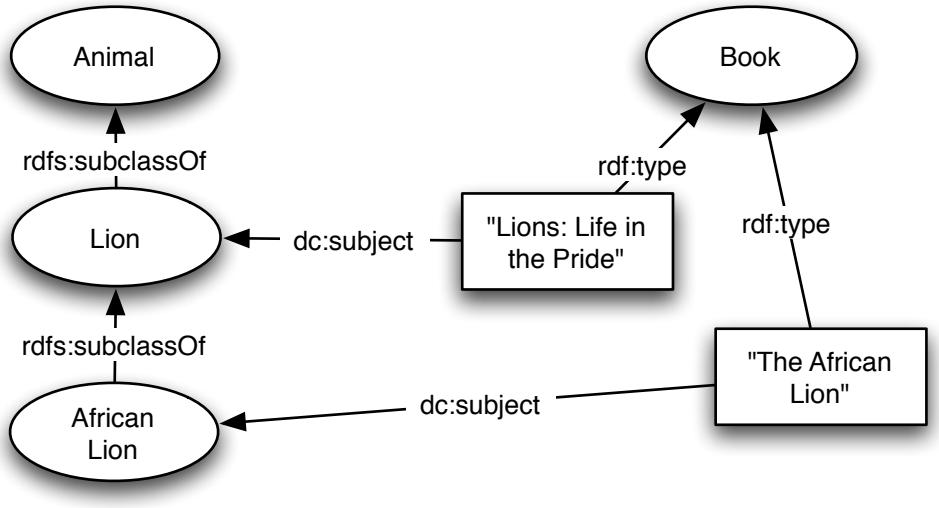

Figure 4.1: Class-as-property-value pattern one

[1] *OWL Full* refers to the complete OWL language. *OWL Lite* and *OWL DL* are subsets of OWL Full that are (in contrast to OWL Full) decidable.

In [NUW05] different patterns to express property relations to classes are presented. In addition to the simple case of using classes directly as values (Figure 4.1), four other patterns, to express such relations in OWL DL or OWL Lite, are described.

The different approaches require additional class or individual definitions in the ontology. Such additional efforts can be automated by complex composition operators. We show the realization of one of the four patterns as a complex composition operator.

Listing 4.6 defines an ontology component (based on the example in [NUW05]) in an N3 based reuse language. Instead of filling the `dc:subject` property with concrete classes, the ontology component declares slots.

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix dc:      <http://purl.org/dc/elements/1.1/> .
@prefix :        <> .

:Book
      a               owl:Class ;
      rdfs:subClassOf owl:Thing .
:BookAboutAnimals
      a               owl:Class .

:bookTitle
      a               owl:DatatypeProperty ;
      rdfs:domain     :Book ;
      rdfs:range      <http://www.w3.org/2001/XMLSchema#string> .

dc:subject
      a               owl:ObjectProperty ;
      rdfs:domain     :Book .

:Animal
      a               owl:Class .

:Lion
      a               owl:Class ;
      rdfs:subClassOf :Animal .

:AfricanLion
      a               owl:Class ;
      rdfs:subClassOf :Lion .

:LionsLifeInThePrideBook
      a               :BookAboutAnimals ;
      rdfs:seeAlso    <http://isbn.nu/0736809643> ;
      :bookTitle      "Lions: Life in the Pride" ;
      dc:subject      << lionSlot >> .

:TheAfricanLionBook
      a               :BookAboutAnimals ;
      rdfs:seeAlso    <http://isbn.nu/089686328X> ;
      :bookTitle      "The African Lion" ;
      dc:subject      << africanLionSlot >> .

rdfs:subClassOf
      a               owl:ObjectProperty .
```

Listing 4.6: Ontology component with slots

There are two individuals — `LionsLifeInThePrideBook` (Line 33) and `TheAfrican LionBook` (Line 39) — as instances of the class `BookAboutAnimals` (Line 10). The first represents a book about lions, the second a book about african lions. In the ontology, the class `Lion` (Line 25) represents the concept of lions, while the concept of african lions is expressed through the class `AfricanLion` (Line 29). Instead of using a class (`Lion` or `AfricanLion`) directly as the value of the `dc:subject` property in the two book individuals, the slots `<<lionSlot>>` (Line 37) and `<<africanLionSlot>>` (Line 43) are defined.

To acquire a complete OWL Full conformant ontology, it is sufficient to bind the corresponding class names to the slots. A composer encapsulating this functionality looks simple (Listing 4.7) and the corresponding composition program is straight forward (Listing 4.8).

```
1 define composer n3pcl.ClassAsPropertyValue1(ontology, valueSlot, valueClassName) {
2     bind ->valueSlot on ->ontology with valueClassName;
3 }
```

Listing 4.7: First pattern composer to represent classes as property values

```
1 pattern composition
2
3 fragment n3.N3Doc lionBooks = /books2.rn3;
4
5 patternCAPV1(ontology=lionBooks, valuepos=lionSlot       , class=:Lion);
6 patternCAPV1(ontology=lionBooks, valuepos=africanLionSlot, class=:AfricanLion);
7
8 print lionBooks to /lionBooks.n3;
```

Listing 4.8: Composition program

The *N3 pattern compositions language* (N3PCL) — used in Listing 4.8 — is a simple extension of the basic composition language. It adopts all its functionalities and adds constructs for the newly defined pattern composers. One of these constructs is `ClassAsPropertyValue1` identified by the keyword `patternCAPV1` that calls the composer from Listing 4.7. The composer takes three arguments: the ontology containing the property value slot, the slot name, and the concerned class.

Now, if we desire to use the ontology component from Listing 4.6 in an OWL DL (or OWL Lite) ontology the composition needs to be adjusted. We apply the second pattern from [NUW05] shown in Figure 4.2.

The idea is to use one specific individual (i.e., an instance of a class) that represents the class itself as property value[2]. Rather than introducing this individual manually, it can be derived by the composition operator presented in Listing 4.9. By replacing the execution of `patternCAPV1` in Listing 4.8 with `patternCAPV2` (Listing 4.10), the composition results in an OWL DL compliant ontology (Listing 4.11).

[2] Advantages and disadvantages of this approach and others are discussed in [NUW05].

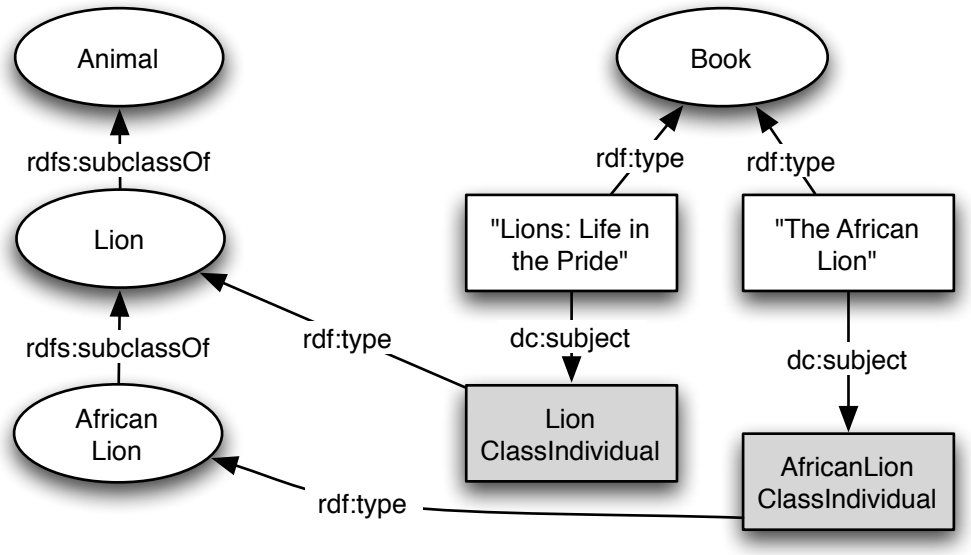

Figure 4.2: Class-as-property-value pattern two

```
1  define composer n3pcl.ClassAsPropertyValue2(ontology, valueSlot, valueClassName) {
2      fragment n3.Resource  individualName  = ':' + valueClassName + 'ClassIndividual'.n3;
3      fragment n3.Triple    individual      = '<<individualName>> a <<className>> .'.rn3;
4
5      bind individualName on individual with individualName;
6      bind className      on individual with valueClassName;
7
8      bind ->valueSlot on ->ontology with individualName;
9
10     bind ontology.subFragment with 'statements[-1]'.fcp;
11     append ->ontology with individual;
12 }
```

Listing 4.9: Second pattern composer to represent classes as property values

The composer creates two new fragments: one for the name of the new individual (Line 2) and one to actual define it as an instance of the corresponding class (Line 3). The latter contains slots for the class and individual names, which are bound to the concrete names (Lines 5 and 6). The slot for the property value is now bound to the individual rather than the class itself (Line 8). At last, the individual definition is appended to the end of the statement list of the ontology (Lines 10 and 11).

```
1  pattern composition
2
3  fragment n3.N3Doc lionBooks = /books2.rn3;
4
5  patternCAPV2(ontology=lionBooks, valuepos=lionSlot       , class=:Lion);
6  patternCAPV2(ontology=lionBooks, valuepos=africanLionSlot, class=:AfricanLion);
7
8  print lionBooks to /lionBooks.n3;
```

Listing 4.10: Composition program

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix dc:      <http://purl.org/dc/elements/1.1/> .
@prefix :        <> .

:Book
      a               owl:Class ;
      rdfs:subClassOf owl:Thing .

:BookAboutAnimals
      a               owl:Class .

:bookTitle
      a               owl:DatatypeProperty ;
      rdfs:domain     :Book ;
      rdfs:range      <http://www.w3.org/2001/XMLSchema#string> .

dc:subject
      a               owl:ObjectProperty ;
      rdfs:domain     :Book .

:Animal
      a               owl:Class .

:Lion
      a               owl:Class ;
      rdfs:subClassOf :Animal .

:AfricanLion
      a               owl:Class ;
      rdfs:subClassOf :Lion .

:LionsLifeInThePrideBook
      a               :BookAboutAnimals ;
      rdfs:seeAlso    <http://isbn.nu/0736809643> ;
      :bookTitle      "Lions: Life in the Pride" ;
      dc:subject      :LionClassIndividual .

:TheAfricanLionBook
      a               :BookAboutAnimals ;
      rdfs:seeAlso    <http://isbn.nu/089686328X> ;
      :bookTitle      "The African Lion" ;
      dc:subject      :AfricanLionClassIndividual .

rdfs:subClassOf
      a               owl:ObjectProperty .

rdfs:subClassOf
      a               owl:ObjectProperty .

:LionClassIndividual
      a               :Lion .

:AfricanLionClassIndividual
      a               :AfricanLion .
```

Listing 4.11: Result of pattern composition

The presented pattern might not be the best solution to solve the problem of classes as property values in OWL DL ontologies. Even if it is a good solution to the human reader, reasoners not knowing about the pattern do not understand the relation between an individual representing a class and the class itself. Sometimes, the use of other patterns presented in [NUW05] might yield a more sophisticated solution.

Therefore, additionally to the automatization of pattern application encapsulated in a composer, the exchangeability of composers implementing different patterns, while solving the same problem, poses a great asset. If an ontology developer is not sure which version of a pattern he likes to apply he can leave it open by defining an ontology component with slots and compose it to a full ontology later using the desired composer.

4.5 Conclusion

This chapter demonstrated initial ideas of ontology compositions that are realizable through invasive composition systems. Since relationships to the composition approaches studied in Chapter 3 were harder to identify than expected, more intense studies in this direction are considered as future work.

The application of the composition tool to define ontology specific composers based on (technology independent) ontology structuring ideas underscores the language independence of the tool. Further research on and experimentation with ontology composition methods will be well supported by the tool.

5 Composition in Query and Transformation Languages

We continue to utilize complex composition operators with Semantic Web languages in this chapter. Beside ontology languages, query languages play an important role in the Semantic Web. While the former bring support for modularity and module management by nature (Section 4.2), the latter might completely lack such capabilities.

One such query languages is Xcerpt [SB04, F$^+$04, F$^+$06] currently under development. Naturally, Xcerpt's developers concentrated on the core functionality first — rule based querying on structured data — and did not yet investigate issues of modularization and composition. Consequently, the current Xcerpt specification [F$^+$06] and its interpreter [SB$^+$06] lack support for composition features — which would enhance Xcerpt's usability.

Xcerpt is thus an excellent candidate to present the applicability of the results of this thesis. The lack of composition support in the language gives room to demonstrate how composition capabilities that exist in many languages can be added to Xcerpt through invasive composition. Through this example, several important ideas are demonstrated.

Firstly, it proves that composition can be designed separately from other functionalities of a language. This leads to a better language structure and that in turn simplifies tool development. Secondly, it shows that the processing of compositions can be completely separated from the interpretation of program logic. Thirdly, it exemplifies benefits for tool development. Since tooling to execute Xcerpt compositions can be generated with the composition tool, the developers might completely discard manual tool building for the new features. Or, if they prefer to integrate modularization in the Xcerpt interpreter itself at some point, they can use generated tooling first to identify which composition approach they finally would like to integrate.

To support the understanding of examples in later sections, Section 5.1 gives an overview of the basic Xcerpt features. Considerations about transferring composition approaches from Chapter 3 to Xcerpt are discussed in Section 5.2. It leads to Section 5.3, which applies the weaving composer (Section 3.3.2) to Xcerpt components, and Section 5.4, which defines an Xcerpt specific composition operator.

5.1 Xcerpt: A Short Introduction

To start with, different concrete syntaxes exist for Xcerpt. We use the so called *term syntax* here that is the best choice when it comes to human readability.

Xcerpt programs work on structured data represented as *data terms*. Data terms can be directly defined in Xcerpt programs, retrieved from XML files, or constructed by Xcerpt rules.

Data terms corresponds directly to XML fragments. Figure 5.1 shows some examples by comparing both notations: The equivalent to an XML element is annotated using square brackets in the term syntax (Example 1). Nested tags are separated by comma (Example 2). Attributes are defined with the keyword **attributes** and are handled like a nested tag (Example 3).

	term syntax	XML syntax
Example 1	`name [ "content" ]`	`<name>content</name>`

Example 2	`tag [`
	`  nestedTag1 [],`
	`  nestedTag2 []`
	`]`
		`<tag>`
		`  <nestedTag1/>`
		`  <nestedTag2/>`
		`</tag>`

Example 3	`tag [`
	`  attributes [`
	`  att1 ["b"],`
	`  att2 ["c"]`
	`  ],`
	`  nestedTag []`
	`]`
		`<tag`
		
		`  att1="b"`
		`  att2="c"`
		`  >`
		`  <nestedTag/>`
		`</tag>`

Figure 5.1: Xcerpt data term syntax in comparison to XML

To query data terms in Xcerpt programs, one can define *query terms*. Such terms are patterns, which in turn correspond to partial data terms. More concretely, they are partial definitions of the data that is queried. A query term matches all the data terms it partially defines.

To define query terms, the syntax for data terms is reused and extended. Some examples are shown in Figure 5.2. Additionally to fixed structures, double square brackets (`[[ ]]`) identify partial but ordered lists of nested tags (Example 1). Braces on the other hand stand for unordered lists. Those can be complete or partial as well — braces (`{ }` — Example 2) or double braces (`{{ }}` — Example 3). To ignore the depth of a nested tag, the **desc** keyword is used (Example 4). Several query terms may exist in a query. They can be combined by the connectives **and** and **or** (Example 5).

	query term	explanation

Example 1	`tag [[`
	`  nestedTag1 [],`
	`  nestedTag2 []`
	`]]`
		Both `nestedTag1` and `nestedTag2` have to be contained in `tag`. Other nested tags may exist but `nestedTag1` has to appear before `nestedTag2`.

Example 2	`tag {`
	`  nestedTag1 [],`
	`  nestedTag2 []`
	`}`
		Both `nestedTag1` and `nestedTag2` have to be contained in `tag`. Other nested tags are not allowed but the relative position of `nestedTag1` and `nestedTag2` is irrelevant.

Example 3	`tag {{`
	`  nestedTag1 [],`
	`  nestedTag2 []`
	`}}`
		Both `nestedTag1` and `nestedTag2` have to be contained in `tag`. Other nested tags may exist. The relative position of `nestedTag1` and `nestedTag2` is irrelevant.

Example 4	`tag [`
	`  desc nestedTag []`
	`]`
		`nestedTag` may be nested in another tag nested in `tag`.

Example 5	`or {`
	`  tag [`
	`  nestedTag1 []`
	`  ],`
	`  tag [`
	`  nestedTag2 []`
	`  ],`
	`}`
		Either `nestedTag1` or `nestedTag2` has to be nested in `tag`.

Example 6	`var X -> tag [ ]`	Binds `tag` to the variable `X`.
Example 7	`tag [ var X ]`	Binds the content of `tag` to the variable `X`.

Example 8	`in {`
	`  resource {`
	`  "f.xml","xml"`
	`  },`
	`  tag [ ]`
	`}`
		Retrieves the data term to match from the file *f.xml*, which is written in *XML* syntax.

Figure 5.2: Xcerpt query term examples

To preserve data from a matched data term, *variable bindings* may be defined in query terms using the **var** keyword. They either bind a variable with a matched substructure (Example 6) or a concrete (string) content of a term (Example 7).

To retrieve data terms from an (XML) file, the **in** keyword is utilized. Inside an **in** "tag" the first nested tag is reserved with the name **resource**. There, the file to load and its concrete syntax are specified (Example 8).

Queries are used inside rules. In addition to the query part, a rule contains a *construct* part. While the query part matches patterns on existing data terms and produces variable bindings, the construct part uses the variable bindings to create new data terms. It consists of *construct terms*, which are defined similar to data terms but may refer to bound variables. Special constructs are *goals* that define the output of an Xcerpt program (i.e., print data to a file or to the standard output stream).

Rules can be chained, meaning that the data constructed by one rule can be used as input to other rules. Several rules might work on same data structures. Rules can be recursive — i.e., access the data terms created in the construct part in the query part of the same rule.

An example of an Xcerpt program that constructs an HTML table from an OWL ontology is presented in Listing 5.2.

```
<RDF>
  <Class ID="Book"/>

  <Class ID="ForChildren"/>

  <DatatypeProperty ID="bookTitle">
    <range  resource="http://www.w3.org/2001/XMLSchema#string"/>
    <domain resource="#Book"/>
  </DatatypeProperty>

  <Book ID="PeterPan">
    <type resource="#ForChildren"/>
    <seeAlso resource="http://isbn.nu/1419151576"/>
    <bookTitle>The Adventures Of Peter Pan</bookTitle>
  </Book>

  <Book ID="TomSawyer">
    <type resource="#ForChildren"/>
    <seeAlso resource="http://isbn.nu/0195114051"/>
    <bookTitle>The Adventures Of Tom Sawyer</bookTitle>
  </Book>

  <Book ID="PippiLongstocking">
    <type resource="#ForChildren"/>
    <seeAlso resource="http://isbn.nu/0670876127"/>
    <bookTitle>The Adventures of Pippi Longstocking</bookTitle>
  </Book>
</RDF>
```

Listing 5.1: Book individuals (simplified OWL RDF/XML syntax)

```
1  GOAL
2     html [
3       head [ title [ "The Book List" ] ],
4       body [
5         h1 [ "The Book List" ],
6         table [
7           tr [
8             td [ b [ "Title" ] ],
9             td [ b [ "ISBN" ] ]
10          ],
11          all tr [
12                td [ var Title ],
13                td [ a [
14                    attributes { href { var ISBN } },
15                    var ISBN
16                ] ]
17              ]
18        ]
19      ]
20    ]
21 FROM
22    bookList [
23      title [ var Title],
24      isbn  [ var ISBN ]
25    ]
26 END
27
28 CONSTRUCT
29    bookList [
30      title [ var Title],
31      isbn  [ var ISBN ]
32    ]
33 FROM
34    in {
35      resource { "file:books1.owl", "xml" },
36      RDF {{
37        Book {{
38          bookTitle [ var Title ],
39          seeAlso [ attributes { resource { var ISBN } } ]
40        }}
41      }}
42    }
43 END
```

Listing 5.2: Xcerpt rules transforming information to a HTML table

The construct rule (Lines 28–43) queries the input file (Listing 5.1) for names and ISBNs of individual books (Lines 34–42) and assembles them in the `bookList` (Lines 29–32). The `bookList` is input (Lines 22–25) to the goal rule (Lines 1–26) that constructs an HTML file containing the information (Lines 2–20).

To learn more about the syntax and capabilities of Xcerpt consult [SB04] and [SB+06].

5.2 Learning from Existing Ideas

Our aim is to reuse ideas from software composition to modularize, reuse, and compose queries and rules. Quite a few composition approaches were introduced in Chapters 2 and 3. It would go beyond the scope of this thesis to investigate the reusability of all these approaches. Therefore, we concentrate on two composition ideas for queries.

Aspect weaving is the first approach we utilize for query composition. The motivation is to use the language independent weaving composer (Section 3.3.2) for aspect weaving of queries without further development on the composer itself. Section 5.3 demonstrates an application of the weaver on an Xcerpt program.

The second approach is the fundamental composition idea of importing modules for reuse as found, for instance, in Java (Section 2.1.1). Such a module system does not yet exist for Xcerpt. In Section 5.4 we develop a complex composition operator to implement such a system.

Certainly, there is more to learn from software composition from which query and rule modularization can profit. The work in this area should be continued.

5.3 Weaving Rules

To apply the weaving composer to Xcerpt rules, concerns that cross-cut rules have to be identified first. Here, we only demonstrate one simple example to show, similar to the OWL case, that weaving of rules is technically possible and that the language independent weaver can work with yet another kind of language.

We add more rules to the Xcerpt program from Listing 5.2 that query different ontologies for book individuals and extend the `bookList`. Listing 5.3 shows one of those rules.

```
1  CONSTRUCT
2     bookList [ title [ var Title], isbn  [ var ISBN ] ]
3  FROM
4     in {
5       resource { "file:books2.owl", "xml" },
6       RDF {{
7         Book {{
8           bookTitle [ var Title ],
9           seeAlso [ attributes { resource { var ISBN } } ]
10        }}
11      }}
12    }
13 END
```

Listing 5.3: Xcerpt rules querying a second source file

Assume that the input files (e.g., */books2.owl*) contain further book individuals, which are, as opposed to the first input file (*/books1.owl*), not all for children. Suppose in addition that the program shall produce a table containing only books for children.

Unfortunately, the necessary information is not contained in the book list. It was already discarded when querying the different source ontology files. As result, we have to change all the corresponding rules.

In such a case, the weaving composer can be applied to weave the aspect of "select only children books" into all rules that query source files. The weaving is shown in Listing 5.4 where *bookList.xcerpt* contains the affected rules.

```
1  cores:
2    core xcerpt.Program bookTransformation = /bookList.xcerpt;
3
4  weavings:
5    weave xcerpt.ConstructQueryRule in bookTransformation.statements
6            before query.queryTerm.children[0].children[0].children[0].children[0] {
7        'type [ attributes { resource { "#ForChildren" } } ]'.xcerpt
8    }
9
10 output:
11   print bookTransformation to /filteredBookList.xcerpt;
```

Listing 5.4: Weaving of rule aspect

The weaving concerns all `ConstructQueryRules` contained in the program (Line 5). The aspect expresses that only books are selected which contain the `type` tag with the attribute `resource` set to `"#ForChildren"` (Line 7).

The result of the weaving is presented in Listing 5.5. (The unchanged goal rule is omitted.)

```
1  CONSTRUCT
2     bookList [ title [ var Title], isbn  [ var ISBN ] ]
3  FROM
4     in {
5       resource { "file:books1.owl", "xml" },
6       RDF {{
7         Book {{
8           type [ attributes { resource { "#ForChildren" } },
9           bookTitle [ var Title ],
10          seeAlso [ attributes { resource { var ISBN } } ]
11        }}
12      }}
13    }
14 END
15
16 CONSTRUCT
17    bookList [ title [ var Title], isbn  [ var ISBN ] ]
18 FROM
19    in {
20      resource { "file:books2.owl", "xml" },
21      RDF {{
22        Book {{
23          type [ attributes { resource { "#ForChildren" } },
24          bookTitle [ var Title ],
25          seeAlso [ attributes { resource { var ISBN } } ]
26        }}
27      }}
28    }
29 END
```

Listing 5.5: Rule weaving result

5.4 Xcerpt Specific Complex Composition Operator: Import

Many programming languages include a *module systems* to import modules and manage them in libraries. Xcerpt does not yet provide such a system.

The following example demonstrates the usefulness of a module system for Xcerpt. First of all, rules can be applied for simple reasoning on ontologies [F+04]. Consider, for instance, the transitive *subClassOf* relationship. In Listing 5.6 different classes and subclasses are defined. The class `BookAboutAnimals`, for example, is a subclass of `Book`. In turn, `BookForChildrenAboutAnimals` is a subclass of `BookAboutAnimals`. That makes `BookForChildrenAboutAnimals` a subclass of `Book` as well (Figure 5.3).

```
<RDF>
  <Class ID="Book"/>

  <Class ID="ForChildren"/>

  <Class ID="BookAboutAnimals">
    <subClassOf resource="#Book"/>
  </Class>

  <Class ID="BookForChildrenAboutAnimals"/>
    <subClassOf resource="#BookAboutAnimals"/>
    <subClassOf resource="#ForChildren"/>
  </Class>
</RDF>
```

Listing 5.6: Book classes (simplified OWL RDF/XML syntax)

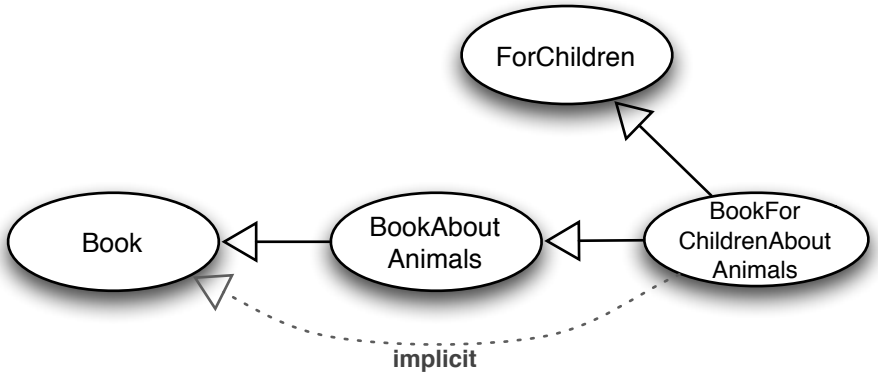

Figure 5.3: SubClassOf relationships

If we write a query that extracts all pairs of sub- and superclasses from the ontology, the pair `BookForChildrenAboutAnimals` / `Book` is not included in the result. However, a rule can be written that performs inference on the subClassOf relationship to make implicit relationships explicit (Listing 5.7).

```
CONSTRUCT
   subClassOf [ var Subclass , var Superclass ]
FROM
   or {
     declaredSubClassOf [  var Subclass -> Class {{ }}, var Superclass -> Class {{ }} ],
     and {
       declaredSubClassOf [ var Subclass -> Class {{ }}, var Z -> Class {{ }}],
       declaredSubClassOf [ var Z, var Superclass -> Class {{ }}]
     }
   }
END

CONSTRUCT
   declaredSubClassOf [ var Subclass , var Superclass ]
FROM
   and {
     <<input>> {{
       desc var Subclass -> Class {{
         subClassOf {{
           attributes {{ Resource { var SuperClassId } }}
         }}
       }}
     }},
     <<input>> {{
       desc var Superclass -> Class {{
         attributes {{ ID { var SuperClassId } }}
       }}
     }}
   }
END
```

Listing 5.7: SubClassOf inference module

5.4.1 Module System for Xcerpt

The rule presented above is reusable in many situations and should thus be extracted into a module. To realize this, we define a module system for Xcerpt.

A module is simply an Xcerpt program, which is stored in a file on a file system[1]. The module library is then a directory tree containing Xcerpt programs.

Additionally, we require means to import these modules into other programs. Such can be provided by an invasive composition system build on Xcerpt as core language. In this composition system, a complex composition operator to import modules needs to be defined. Listing 5.8 contains a possible specification.

[1] This is to keep the example simple. As an alternative to a file path, global URIs could be used and modules could be fetched from the web. This is certainly desirable in the case of Xcerpt modules.

```
1 define composer reusexcerpt.ImportComposer(moduleLocation, ruleName, args) {
2     fragment xcerpt.Program module = ->moduleLocation;
3
4     foreach(element : args) {
5         bind ->element.slot on module with element.value;
6     }
7
8     return module.statements;
9 }
```

Listing 5.8: Import composer

The composer's arguments include the `location` (file path) of the module and the `name` (i.e., top-level construct term) of the rule inside the module that should be imported. Furthermore, the `args` argument points at slots in the module and specifies values to bind them with. Slots in modules are convenient since modules work on data structures provided by the importing program not known at module specification time. The module should thus be configurable to a certain extent. After the module is loaded (Line 2) all indicated slots are bound (Line 4-6). Finally, the rules are returned (Line 8) to be bound in-place in the calling program. Note that not only the requested but all rules contained in the module are returned since they may depend on each other.

The import composer and the inference module utilize a reuse language. This language includes several constructs for slot declaration and one to call the import composer. The latter may be used in an Xcerpt program as an alternative to rule specifications.

The program that imports the inference module from Listing 5.7 — to construct a complete list of sub- and superclasses based on the ontology in Listing 5.6 — is presented in Listing 5.9.

```
1  IMPORT
2     /inference/subClassOf.rxcerpt::subClassOf [
3        input = classDefs
4     ]
5  END
6
7  GOAL
8     result [ all var X ]
9  FROM
10    var X -> subClassOf {{ }}
11 END
12
13 CONSTRUCT
14    classDefs [ all var Class ]
15 FROM
16    in {
17      resource { "file:bookClasses.owl", "xml" },
18      RDF {{
19        desc var Class -> Class {{ }}
20      }}
21    }
22 END
```

Listing 5.9: Xcerpt program with import

5.4.2 Refinement of the Module System: Information Hiding

The import composer defined works in a rather simple fashion. It takes all rules from one program (the module) and puts them into another one. Hereby, all rules from the module are imported. This has two main disadvantages. Firstly, all rules of the module become available in the program, despite the fact that there is only one rule specified to be imported. Consequently, the program can access rules that are not directly imported, which breaks information hiding. Secondly, the situation might break the whole program since the rules imported without knowledge might use similar names as the program.

Sometimes, modules contain rules that should not be importable. This is the case for rules in a rule chain that do not deliver a final result but only prepare data for other rules. The second rule in the inference module is such a rule. Therefore, another desired feature is to specify in a module which rules are importable and which are not.

We refine the reuse language and the import composer to realize the additional feature and to solve the problems mentioned above. At first, we integrate a new type of rule into the reuse language — the modified rule. It wraps an ordinary Xcerpt rule (defined in the core language) and adds a modifier to it. The modifier can be `PUPLIC`, marking the rule as importable, or `PRIVATE`, to hide the rule to the outside. Furthermore, we determine that rules without modifier are considered private. The rules in the inference module are extended with modifiers in Listing 5.10.

```
1  PUBLIC CONSTRUCT
2     subClassOf [ var Subclass , var Superclass ]
3  FROM
4     or {
5       declaredSubClassOf [  var Subclass -> Class {{ }}, var Superclass -> Class {{ }} ],
6       and {
7         declaredSubClassOf [ var Subclass -> Class {{ }}, var Z -> Class {{ }}],
8         declaredSubClassOf [ var Z, var Superclass -> Class {{ }}]
9       }
10    }
11 END
12
13 PRIVATE CONSTRUCT
14    declaredSubClassOf [ var Subclass , var Superclass ]
15 FROM
16    and {
17      <<input>> {{
18        desc var Subclass -> Class {{
19          subClassOf {{
20            attributes {{ Resource { var SuperClassId } }}
21          }}
22        }}
23      }},
24      <<input>> {{
25        desc var Superclass -> Class {{
26          attributes {{ ID { var SuperClassId } }}
27        }}
28      }}
29    }
30 END
```

Listing 5.10: SubClassOf inference component with public and private rules

The next step is to solve the problem of information hiding by refining the import composer. Since we can only solve the import with invasive composition in the tool, we can not omit the inclusion of not imported and private rules into the importing program. However, we are able to hide them inside. Under the assumption that a constructed data term can only be accessed via its top-level name[2], we nest the top-level term inside another (new top-level) term with an unique label that does not appear in the original program. Now the original top-level term is hidden. In addition to wrapping a top-level construct term, all queries that access this term have to be modified to take the wrapping into account as well. The second version of the import composer (Listing 5.11) implements exactly this functionality.

```
1  define composer reusexcerpt.ImportComposer(moduleLocation, ruleName, args) {
2      fragment xcerpt.Program   module = ->moduleLocation;
3
4      foreach(r : module.statements) {
5          if (r instanceof reusexcerpt.ModifiedConstruct
6                  && r.modifier instanceof reusexcerpt.PublicModifier
7                  && r.construct.construct.localSpec.label equals ruleName) {
8
9              bind r with r.construct;
10         }
11         else {
12             if (r instanceof reusexcerpt.ModifiedConstruct) {
13                 bind r with r.construct;
14                 fragment xcerpt.ConstructQueryRule r = r.construct;
15             }
16
17             fragment xcerpt.NCName        secretName = ? .xcerpt;
18
19             fragment xcerpt.ConstructTerm ct         = r.construct;
20             fragment xcerpt.ConstructTerm ctWrapper  = '<<secretName>> [ <<hiddenCt>> ]'.rxcerpt;
21
22             bind hiddenCt   on ctWrapper with ct;
23             bind secretName on ctWrapper with secretName;
24             bind ct                      with ctWrapper;
25
26             foreach(rule : module.statements) {
27                 traverse(qt : rule.query) {
28                     if (qt instanceof xcerpt.StructuredQt) {
29                         if (qt.localSpec.label equals ct.localSpec.label) {
30                           fragment xcerpt.QueryTerm qtWrapper = '<<secretName>>  [ <<hiddenQt>> ]'.rxcerpt;
31
32                           bind hiddenQt   on qtWrapper with qt;
33                           bind secretName on qtWrapper with secretName;
34                           bind qt                      with qtWrapper;
35                         }
36                         prune;
37                     }
38                 }
39             }
40         }
41     }
42
```

[2] In fact, the Xcerpt specification allows to use the **desc** operation on top-level query terms. Thus, data can be accessed without knowing the name of a top-level data term. For our import solution, we assume the availability of the **desc** operations only for non-top-level terms.

```
43      foreach(element : args) {
44          bind ->element.slot on module with element.value;
45      }
46
47      return module.statements;
48  }
```

Listing 5.11: Import composer version two

After loading the module (Line 2) and before binding its slots (Lines 43-45), the composer executes the following new functionality: For each rule it is first checked if it is public (i.e., `ModfiedConstruct` with modifier `PUBLIC`) and if it is the imported rule (Lines 5–7). If it is indeed the imported one, the modified rule is replaced with the contained core language rule (Line 9). In all other cases the rule needs to be hidden. If it is a modified rule (`PRIVATE` or `PUBLIC` but not imported) the modifier is removed (Lines 12–15). Next, a fragment with an unique name (`?`[3]) is defined (Line 17). The (top-level) `ConstructTerm` is wrapped in a `ConstructTerm` with the hidden name and then replaced by it (Lines 19–24). Afterwards, the content of all rules is traversed and the top-level `QueryTerm`s are identified (Lines 26–28). If a `QueryTerm` accesses the hidden construct, it is wrapped as well (Lines 29–35). In the end, the imported and the hidden rules are returned (Line 47).

Listing 5.12 displays the result of the composition from Listing 5.9 with the refined import composer.

```
1  CONSTRUCT
2    subClassOf [ var Subclass , var Superclass ]
3  FROM
4    or {
5      n103873322 [
6       declaredSubClassOf [ var Subclass -> Class {{ }} , var Superclass -> Class {{ }} ]
7      ],
8      and {
9        n103873322 [
10         declaredSubClassOf [ var Subclass -> Class {{ }} , var Z -> Class {{ }} ]
11       ],
12       n103873322 [
13         declaredSubClassOf [ var Z , var Superclass -> Class {{ }} ]
14       ]
15     }
16   }
17 END
18
19 CONSTRUCT
20   n103873322 [
21     declaredSubClassOf [ var Subclass , var Superclass ]
22   ]
```

[3] special composition language interpreter that randomly produces a cryptic string

```
23 FROM
24   and {
25     classDefs {{
26       desc var Subclass -> Class {{
27         subClassOf {{
28           attributes {{ Resource { var SuperClassId } }}
29         }}
30       }}
31     }},
32     classDefs {{
33       desc var Superclass -> Class {{
34         attributes {{ ID { var SuperClassId } }}
35       }}
36     }}
37   }
38 END
39
40 GOAL
41     result [ all var x ]
42 FROM
43     var x -> subClassOf {{ }}
44 END
45
46 CONSTRUCT
47   classDefs [ all var x ]
48 FROM
49   in {
50     resource { "file:wine.owl", "xml" },
51     RDF {{
52       desc var x -> Class {{ }}
53     }}
54   }
55 END
```

Listing 5.12: Import composition result

5.5 Conclusion

We demonstrated how invasive composition systems can be applied to add well-known composition capabilities to newly developed languages. Xcerpt is a good candidate for further experimentation. It is a Semantic Web language that should be enriched with more composition capabilities to gain better usability.

As indicated, the composition tool now has the capabilities to be applied for experimentation. Thus, it should be propagated to Xcerpt developers to support the maturation of the language.

6 Conclusion

After developing and evaluating complex composition operators in the last chapters, it is time to draw conclusions about our work. At first, we exemplify our view on how the results of this thesis can contribute to future development of E-CoMoGen (Section 6.1) and further research on composition in Semantic Web languages (Section 6.2). We end by emphasizing the major achievement of our work in Section 6.3.

6.1 Future Work on Composition Tooling

In this thesis, we presented the extended tool architecture as a means to implement interpretation of composition — and composer definition — languages. It was utilized for the *basic composition language* of E-CoMoGen (Appendix A). This language's design, however, is solely based on the requirements of our examples and composition operators. It does not claim to be the future composition language of E-CoMoGen. Development of a richer one should be initiated soon. That language should be well-structured by defining a core and several additional modules. The modules can then be reused as needed in specialized composition languages — like the weaving language — and reuse languages (for in-place operations). Consequently, a library of composition language modules and composition languages should be realized.

Another goal should be the development of a library of composition operators. As by now, no central registry or anything alike exists to reuse composers (without copying the actual definitions). This task is connected to the composition language library requirement since composition operators are part of a specialized composition language.

Concerning the language independence of composers, one should as well think about a library of modular (partial) language metamodels. Such metamodels may define concepts common to a set of languages (e.g., concepts found in all ontology languages). One can then define composers that are not completely language independent but independent within one language domain (e.g., that work on fragments written in any ontology language). This corresponds to the idea of introducing a common metamodel of the slot and hook concepts to enable primitive composers to work with any reuse language.

Other structuring issues concerning the tool have to be addressed as well. In this thesis, the architecture for composition operator integration was defined. Beside that, other parts of the tool should be equally clearly structured. Most importantly, the overall structure of the tool should be clarified.

The mentioned tool parts are, for instance, the *component model generato*r (CoMoGen) and the *composition execution core*. The latter is the heart of the tool that

executes the compositions by model rewriting. It is (part of) the composition language interpreter system. Which parts belong to the interpretation and which part is the fixed core should be clarified. In any case, a real component model generator — implementing the algorithms from Section 2.2.4 — does not exist at the moment. Reuse language extensions are always done manually. Hence, such a generator should be developed.

6.2 Future Work on Ontology and Query Composition

As mentioned in the conclusions of Chapters 4 and 5, many future research possibilities exist in the field of ontology and query composition. The composition tool is now in a position to support experimentations for those investigations. Thus, work should be continued following the results of these chapters.

The investigations done in this thesis exposed the necessity of real-word examples and use cases. Unfortunately, there is a lack of experience with using Semantic Web languages in real life. Most research uses academic toy examples.

A good candidate to derive use cases from is the gene ontology [Con06a]. It is a large ontology and certainly has requirements for structuring that can be met with composition. Contact with domain experts (i.e., people using this ontology) exists. Thus, requirements can be gathered and solutions based on complex composition operators can be proposed.

In the case of Xcerpt, contact with the developers should be intensified. The composition tool is close to a state in which it is deployable to external users. Thus, Xcerpt developers should be emphasized to use it. Here, the import composer poses a good initial idea to present and to discuss.

6.3 Results

The main achievements of our work are twofold: Firstly, we came to important realizations about language independent complex composer development. Secondly, we were able to identify complex composers for two Semantic Web languages, which differ in nature from each other and traditional programming languages.

The results about composition language interpretation and composer definition closed a gap in the theory behind the composition tool. The tool's overall structure became much clearer and its implementation became much more applicable for experimentations with different languages and composition operators.

The complex composers developed in this thesis proved that complex composers can be a combination of primitive composers and that complex composers can (but do not have to) be language independent. We further illustrated how a composition approach can be transferred from software engineering to the Semantic Web domain directly by reusing a language independent complex composer (i.e., the weaving composer).

However, it became clear that language dependent complex composers have to be developed to describe meaningful compositions specific to Semantic Web languages (e.g., the OWL pattern composer or the Xcerpt import composer). With such composers we demonstrated new composition ideas for the Semantic Web domain. The examples show, and can help to convince others, that invasive composition is helpful to develop and to execute composition techniques for ontology and query languages. In the future, this will help to accelerate development of further composition ideas.

As a final remark, we can report without reservation that this thesis is a success. Vital results for — language independent and language dependent — invasive composition applied on Semantic Web languages have been achieved. These are important steps into the future of composition in the Semantic Web.

A Basic Composition Language Reference

This is a reference of the basic composition language included in E-CoMoGen. The language can be used in any invasive composition system to compose fragment components independent of the component language. The abstract and concrete syntax definitions of the language are included in Appendix B. Parts of the language are reused in specialized composition languages. The concrete syntax might vary in such cases.

A.1 Composition Statements

A composition program is a sequence of *composition statements*. The following composition statements are available in the basic composition language:

- **`"fragment" type fragment-name "=" value ";"`**
  declares a fragment. A fragment has a *type* (i.e., the language construct it is an instance of) and a *name*. The *value* that is assigned can be:
  - **`( "/" path-element )+ "." extension`**
    the *path* to a file containing the fragment.
  - **`name`**
    the *name* of another fragment that should be assigned to the new name.
  - **`( "'" code-piece "'" "+" | fragment-name "+" )+ "." extension`**
    the code of a new fragment to construct. The *extension* identifies the concrete syntax the fragment is written in. The fragment code is a concatenation of defined *code pieces* and code pieces that are extracted from other *fragments*. This enables the construction of new fragments from existing ones based on their content only. The types of the existing fragments are irrelevant here.
- **`"print" fragment-name "to" path ";"`**
  prints the *fragment* to the *path* (path definition is equal to path definition in fragment declarations).
- **`"if" condition "{" if-body "}" ( "else" "{" else-body "}" )?`**
  evaluates the condition and executes either the composition statements contained in the *if-body* or in the *else-body*. The condition can be:
  - **`fragment-name1 "equals" fragment-name2`**
    that types and code of two *fragments* are equal.

  - **fragment-name "instanceof" type**
    that a *fragment* is an instance of a certain *type* (i.e., construct).
  - **"isbound" slot "on" fragment-name**
    that a slot is bound (i.e., does not exist anymore) on a *fragment*.

- **"foreach" "(" fragment-name ":" fragment-list ")" "{" body "}"**
  iterates over the *fragment list* and executes the *body* for every iteration. In every iteration, the current fragment in the list is made available under the given *fragment name*.

- **"traverse" "(" fragment-name ":" fragment-list ")" "{" body "}"**
  traverses over the tree of sub-fragments of the fragments in the *fragment list* and executes the *body* for each sub-fragment. In every step, the current sub-fragment is made available under the given *fragment name*. The traversing of a sub-tree can be prevented by using the **prune;** statement.

A.2 Composer Statements

Composer statements are special composition statements that execute composition operators. The composition language includes such statements for the primitive composition operators. Their usage corresponds to the definition of the primitive composers in Section 2.3.1.

- **"bind" (slot-name "on")? fragment-name "with" value ;**
  queries for the *slot(s)* in the fragment and binds them with the *value*. The value is a fragment and the ways of specifying it correspond to the ways of defining the value of a *fragment declaration*. If no slot name is specified, the *fragment* is replaced with the *value*.

- **"extend" (hook-name "on")? fragment-name "with" value ";"**
  queries for the *hook(s)* in the *fragment* and extends them with the *value*. If no hook name is specified, the *value* is added to the list containing the *fragment*.

- **"prepend" (hook-name "on")? fragment-name "with" value ";"**
  queries for the *hook(s)* in the *fragment* and prepends the *value* (specialization of `extend`).

- **"append" (hook-name "on")? fragment-name "with" value ";"**
  queries for the *hook(s)* in the *fragment* and appends the *value* (specialization of `extend`).

A.3 Modularization and Composition

Composition programs are themselves composable. In particular, they may contain in-place bind operations that are especially applicable in composer definitions (see next section).

- **"->" value**
  Binds the *value* in-place.

In-place binds are available for values, fragment names, fragment types, and variation point names.

A.4 Composer Definitions

The composer definition language inherits all functionality from the composition language. In addition, it includes a construct to define complex composers. For each composer defined, a construct to call the composer has to exist in a specialized composition language.

- **"define" "composer" construct "(" arguments ",")* ")"**
  "{" statementList ("return" returnFragment ";")? "}"
  defines the composer for the *construct* utilized to call it. The *arguments* have to correspond to the references of the call construct. The *statement list* may contain any composition statement from the basic composition language. The *return fragment*, if specified, is bound in-place after the composer's execution.

The arguments are accessible as fragments. Thus, they can be directly composed, meaning that the fragments passed to the composer are bound and extended with each other. However, often arguments configure the composer itself. In this case, they need to be bound in-place (e.g., one argument is the name of a slot the composer should address: `bind ->slotNameArg on fragArg1 with fragArg2;`).

B Language Grammars and Metamodels

This is a collection of language descriptions for all languages used in this thesis. For each language, there are two commutable descriptions of the abstract syntax — a grammar and a metamodel in UML notation. In addition, one concrete syntax grammar is defined for each language. The two grammar languages are explained in short. The metamodels are annotated in standard UML class diagram notation.

The *abstract syntax definition language* is oriented at the one used in [Mey90]. There is a bidirectional mapping relationship between an abstract syntax grammar and a metamodel. The composition tool implements this mapping.

In the language, there are four kinds of grammar rules:

- **Aggregation:** `rule-name "=" (ref-name ":" ref-rule card? ",")+ ";"`
  is a rule that is an aggregation of references to other rules. It corresponds to the definition of references in a metamodel.
- **Choice:** `rule-name "=" (alternative-rule "|" )* ";"`
  is a rule defining a set of alternatives. It corresponds to the *subclass-of* relationship in a metamodel where all subclasses (i.e., alternatives) can substitute their superclass.
- **Inheritance:** `rule-name ==> super-rule ";"`
  defines that a rule inherits all properties from a super-rule. It also corresponds to the *subclass-of* relationship in a metamodel. Here, it expresses that a subclass inherits all references from its superclass. Such expressiveness is usually not supported by grammar languages and only available in metamodels. Note that the rules `super-rule = alternative-rule ";"` and `alternative-rule ==> super-rule ";"` technically express the same thing. The distinction exists to express the different usages of *subclass-of* in distinct ways, which enhances readability.
- **Terminal:** `rule-name ";"`
  Empty rules (i.e., classes without references) act as terminal symbols. A special case is the terminal rule `S`, which is predefined. It denotes the infinite set of strings and is mapped to an attribute of the primitive type *String* in metamodels.

It is possible to refer to rules defined in other grammars to extend an existing language:

- `language-name "." rule-name` .

The *concrete syntax definition language* is used to write concrete syntax grammars for a previously defined abstract syntax. For one abstract syntax, different concrete syntaxes may exist. Composition works solely on the abstract syntax. Thus, fragments written in different concrete syntaxes can be composed. The tool can also be used to translate fragments or programs from one concrete syntax into another. A concrete syntax grammar has a header line and rules oriented at EBNF rules [Int96].

- **Header:** `"CONCRETESYNTAX" cs-name "FOR" as-name`
  defines for which abstract syntax this concrete syntax is defined.
- **Rule:** `rule-name "::=" ( element* "|")* ";"`
  The rule name corresponds to the name of an *aggregation* rule in the abstract syntax. The rule can contain nested sequences and alternatives of elements. An element is either part of the concrete syntax, of a concrete string encapsulated in quote symbols (`"` or `'`), or of a reference defined in the abstract syntax. References that have a multiplicity greater than one may be used several times in a concrete syntax definition. They will be merged to one reference in the abstract syntax tree.
- **String Terminals:** `reference "[" regex "]"`
  References to the rule `S` (primitive type String) are enriched with a regular expression to define a set of strings that may occur there. The format of regular expressions corresponds to the one applied in ANTLR grammars [Par06].

Note that a concrete syntax has to be defined for each abstract syntax aggregation rule. This includes referenced rules defined in other abstract syntax grammars. In cases where a concrete syntax for such rules should be reused (e.g., in a reuse language) a simple mechanism to extend concrete syntax grammars is provided.

- **Header:** `"CONCRETESYNTAX" cs-name "FOR" as-name "EXTENDS" super-cs`
  All the definitions that exist in the super-grammar are imported but may be overridden.

In the following, the abstract and concrete syntax definition languages are used to define themselves and each other. *Java-* is a subset of the Java language. The grammars are derived from the Java grammar available from [Par06]. The *Notation 3* grammar is an assimilated version of the grammar available from [Bec06]. While authored based on the language description from [F+06], the *Xcerpt* grammars describe (a subset of) a slightly older version of the language that is supported by the Xcerpt prototype implementation available from [SB+06].

B.1 Abstract Syntax Definition Language

```
1  AbstractSyntax = rules:Rule+;
2
3  Rule           = name:Identifier, definition:Definition?;
4
5  Identifier     = identifier:S;
6
7  Definition     = Aggregation | Choice;
8
9  Choice         = options:Identifier+;
10
11 Aggregation    = reference:Reference+;
12
13 Reference      = name:Identifier, rule:Identifier, cardinality:Cardinality?;
14
15 Cardinality    = PLUS | STAR | QUESTIONMARK;
16
17 PLUS;
18 STAR;
19 QUESTIONMARK;
```

Listing B.1: Abstract syntax grammar of abstract syntax definition language

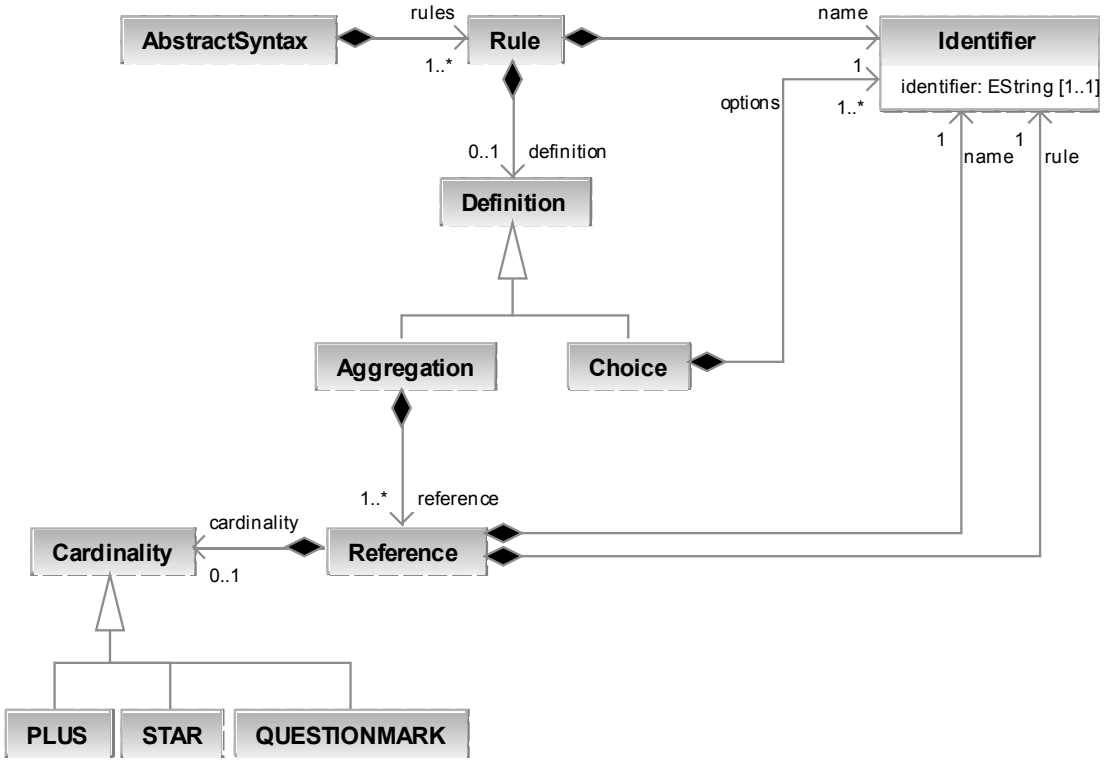

Figure B.1: Abstract syntax metamodel of abstract syntax definition language

```
1  CONCRETESYNTAX as FOR abstractsyntax
2
3  AbstractSyntax ::= rules+;
4
5  Rule           ::= (name ("=" definition)? ";") | (definition "==>" name ";");
6
7  Identifier     ::= identifier[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+];
8
9  Choice         ::= options ( "|" options)* ;
10
11 Aggregation    ::= reference ( "," reference )*;
12
13 Reference      ::= (name ":" rule cardinality?) ;
14
15 PLUS           ::= "+";
16 STAR           ::= "*";
17 QUESTIONMARK   ::= "?";
```

Listing B.2: Concrete syntax grammar of abstract syntax definition language

B.2 Concrete Syntax Definition Language

```
1  ConcreteSyntax = name:S, languageName:S, superGrammar:S?, rules:Rule+;
2
3  Rule           = name:Metaidentifier, definition:Choice;
4
5  Metaidentifier = identifier:S;
6
7  Choice         = options:Aggregation+;
8
9  Aggregation    = parts:SingleDefinition+;
10
11 SingleDefinition = body:DefinitionBody, cardinality:Cardinality?;
12
13 DefinitionBody = SubDefinition | Reference | CsString;
14
15 SubDefinition  = definition:Choice;
16
17 Reference      = name:S, regex:S?;
18
19 CsString       = value:S;
20
21 Cardinality    = PLUS | STAR | QUESTIONMARK;
22
23 PLUS;
24 STAR;
25 QUESTIONMARK;
```

Listing B.3: Abstract syntax grammar of concrete syntax definition language

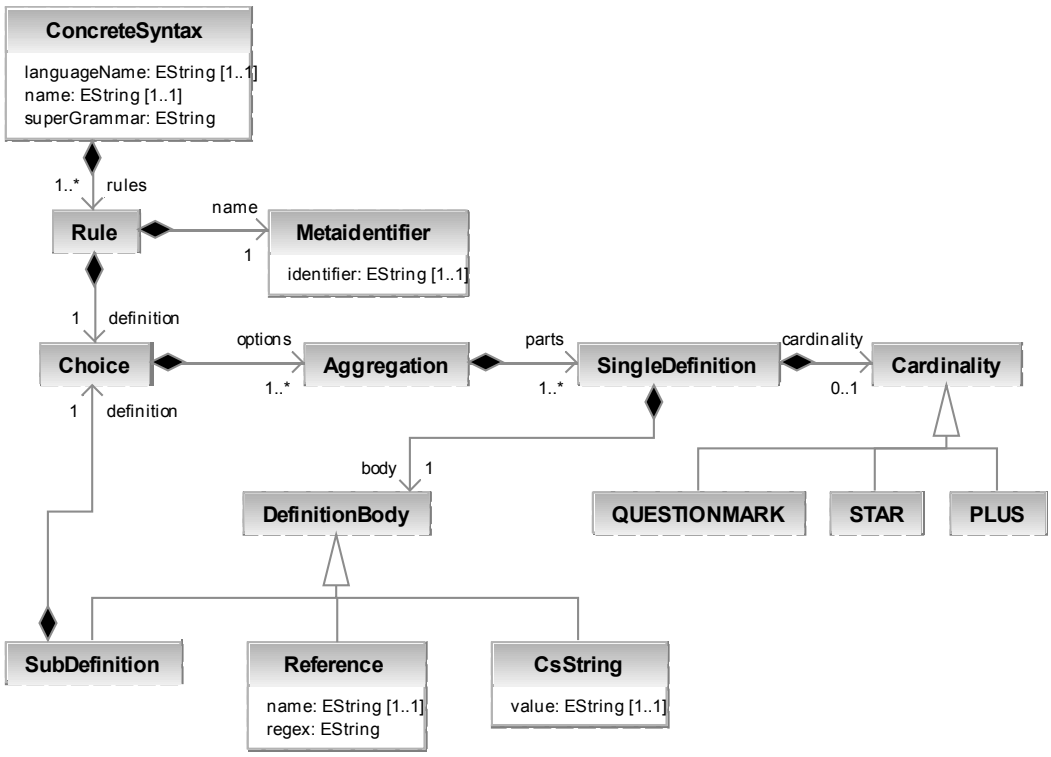

Figure B.2: Abstract syntax metamodel of concrete syntax definition language

```
1  CONCRETESYNTAX cs FOR concretesyntax
2
3  ConcreteSyntax ::= "EBNF" name[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+]
4     "FOR" languageName[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+]
5     "EXTENDS" superGrammar[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+])? rules+;
6
7  Rule           ::= name "::=" definition ";";
8
9  Metaidentifier ::= identifier[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+];
10
11 Aggregation    ::= parts+;
12
13 Choice         ::= options ("|" options)*;
14
15 SingleDefinition ::= body cardinality?;
16
17 Reference      ::= name[(('A'..'Z'|'a'..'z'|'0'..'9')+ '.' )? ('A'..'Z'|'a'..'z'|'0'..'9')+]
18    regex['[' ( ' '|'('|')'|'|'|'?'|'*'|'+'|'~'|'.'| ('\'' ( ~('\''|'\\') | ('\\' .))* ) '\'' )* ']']?;
19
20 CsString       ::= value[('\"' (~'\"')* '\"') | ('\'' (~'\'')* '\'')];
21
22 SubDefinition  ::= "(" definition ")";
23
24 PLUS ::= "+";
25 STAR ::= "*";
26 QUESTIONMARK ::= "?";
```

Listing B.4: Concrete syntax grammar of concrete syntax definition language

B.3 Component Model

```
1  VariationPoint = type:AbstractFragmentType, name:AbstractVariationPointName;
2
3  AbstractFragmentType = FragmentType | FragmentTypeSlot;
4  FragmentType         = language:S, construct:S;
5  FragmentTypeSlot;
6  FragmentTypeSlot   ==> Slot;
7
8  AbstractVariationPointName = VariationPointName | VariationPointNameSlot;
9  VariationPointName         = name:S;
10 VariationPointNameSlot;
11 VariationPointNameSlot   ==> Slot;
12
13 Slot;
14 Slot ==> VariationPoint;
15
16 Hook;
17 Hook ==> VariationPoint;
```

Listing B.5: Grammar of component model concepts

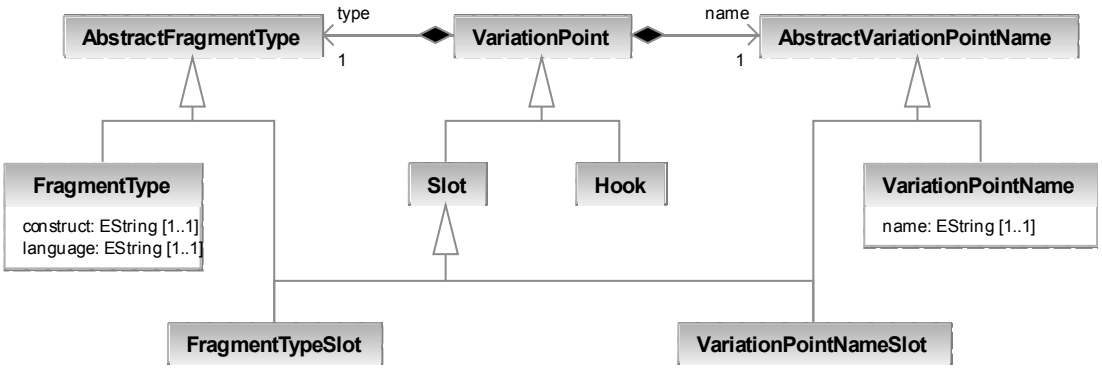

Figure B.3: Metamodel of component model concepts

B.4 Basic Composition Language

```
1  CompositionProgram       = statementList:CompositionStatementList;
2
3  CompositionStatementList = statements:CompositionStatement*;
4
5  CompositionStatement = Composer | FragmentDeclaration | Print | Conditional | ForEachLoop | Traverse|Prune;
6
7  FragmentDeclaration  = type:componentmodel.AbstractFragmentType, name:AbstractFragmentName, value:Value+;
8
9  componentmodel.AbstractFragmentType = FragmentTypeBind;
10 FragmentTypeBind;
11 FragmentTypeBind       ==> Bind;
12
13 AbstractFragmentName     = FragmentName | FragmentNameBind;
14 FragmentNameBind;
15 FragmentNameBind       ==> Bind;
16 FragmentName             = name:S, subFragment:AbstractFragmentName?, index:S?;
17
18 Value                    = Location | AbstractFragmentName | FragmentDefinition | ValueBind;
19 ValueBind;
20 ValueBind               ==> Bind;
21
22 Location                 = path:S;
23
24 FragmentDefinition       = codePieces: CodePiece+, fileExtension:S;
25
26 CodePiece                = ConcreteCodePiece | AbstractFragmentName | RandomPiece;
27 ConcreteCodePiece        = code:S;
28 RandomPiece;
29
30 Print                    = fragment:AbstractFragmentName, location:Location;
31
32 ForEachLoop  = fragmentName:AbstractFragmentName, list:AbstractFragmentName, body:CompositionStatementList;
33 Traverse     = fragmentName:AbstractFragmentName, list:AbstractFragmentName, body:CompositionStatementList;
34 Prune;
35
36 Conditional               = condition:ConcatenatedCondition,
37                               ifBody:CompositionStatementList, elseBody:CompositionStatementList;
38 Condition                = InstanceofCondition | IsBoundCondition | EqualsCondition;
39 ConcatenatedCondition    = conditions:Condition+;
40 InstanceofCondition      = fragment:AbstractFragmentName, type:componentmodel.AbstractFragmentType;
41 IsBoundCondition         = fragment:AbstractFragmentName, slot:componentmodel.AbstractVariationPointName;
42 EqualsCondition          = fragment1:AbstractFragmentName, fragment2:AbstractFragmentName;
43
44 VariationPointNameBind;
45 VariationPointNameBind                  ==> Bind;
46 componentmodel.AbstractVariationPointName = Bind;
47
48 Composer   = Bind | Extend | Prepend | Append;
49
50 Bind       = slot:componentmodel.AbstractVariationPointName?, fragment:AbstractFragmentName?, value:Value;
51
52 Extend     = hook:componentmodel.AbstractVariationPointName?, fragment:AbstractFragmentName?, value:Value+;
53 Prepend;
54 Prepend  ==> Extend;
55 Append;
56 Append   ==> Extend;
```

Listing B.6: Abstract syntax grammar of basic composition language

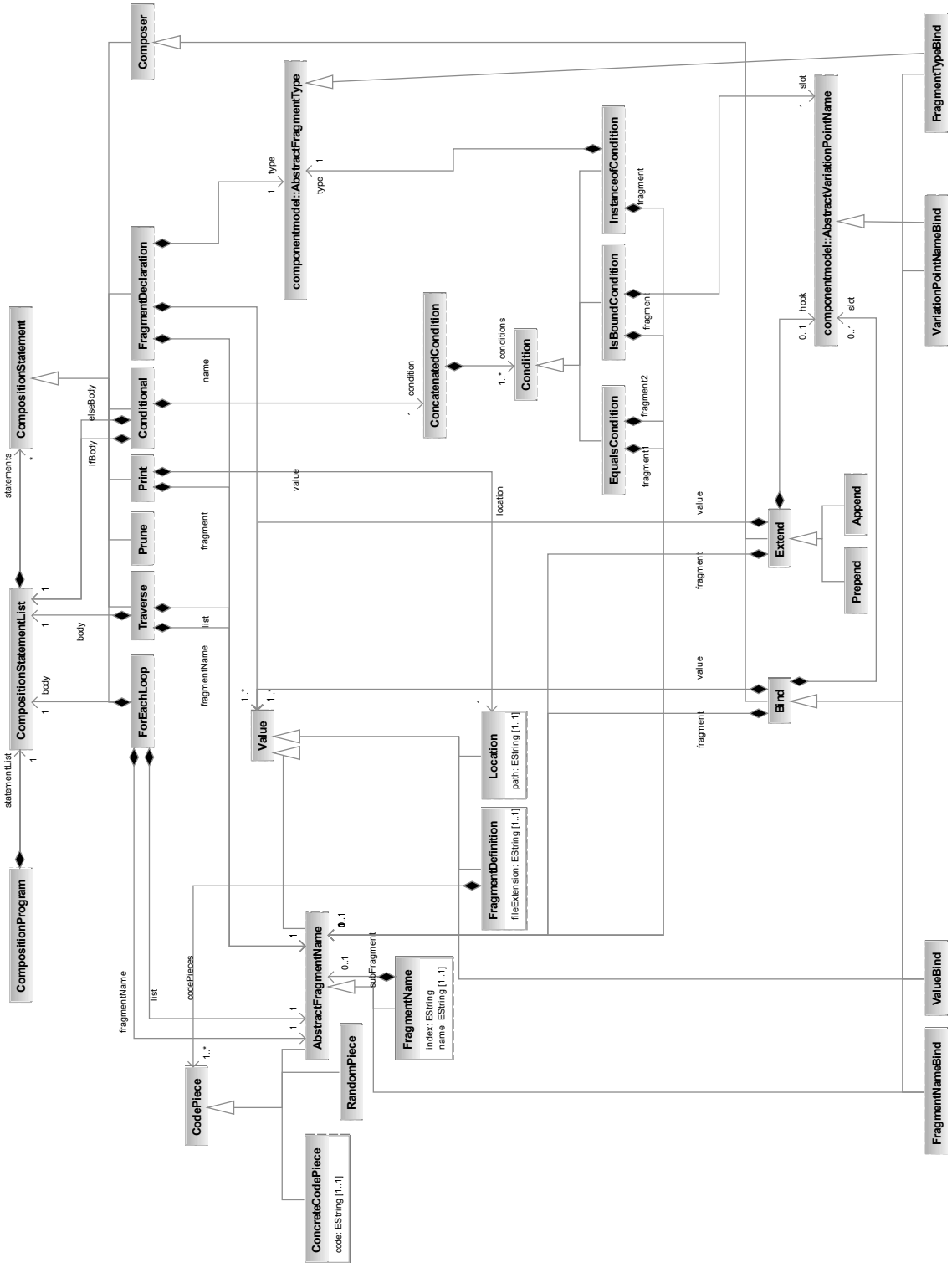

Figure B.4: Abstract syntax metamodel of basic composition language

```
1  CONCRETESYNTAX fcp FOR compositionlanguage
2
3  CompositionProgram       ::= "composition" "program" statementList;
4
5  CompositionStatementList ::= statements*;
6
7  FragmentDeclaration ::=  "fragment" type name "=" value ("," value)* ";";
8
9  Location            ::=  path[('/' ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'.'|'-')+)+];
10
11 FragmentDefinition  ::=  codePieces ('+' codePieces)* "."
12                          fileExtension[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
13
14 ConcreteCodePiece   ::=  code[ '\'' (~('\''))* '\''];
15 RandomPiece         ::=  "?";
16
17 Bind        ::=  "bind"   (slot "on")? fragment "with" value ";";
18
19 Extend      ::=  "extend" (hook "on")? fragment "with" value ("," value)* ";";
20
21 Prepend     ::=  "prepend" (hook "on")? fragment    "with" value ";";
22 Append      ::=  "append"  (hook "on")? fragment  "with" value ";";
23
24 Print       ::=  "print" fragment "to" location ";";
25
26 ValueBind              ::=  "->" value;
27 FragmentNameBind       ::=  "->" value;
28 FragmentTypeBind       ::=  "->" value;
29 VariationPointNameBind ::=  "->" value;
30
31 FragmentName ::=  name[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*]
32                   ("[" index[('-')? ('0'..'9')+] "]")? ("." subFragment)?;
33
34 componentmodel.VariationPointName ::=  name[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
35 componentmodel.FragmentType ::=  (language[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*] ".")?
36                                  construct[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
37
38 Conditional  ::=  "if" "(" condition ")" "{" ifBody "}" ("else" "{" elseBody "}")?;
39 ConcatenatedCondition ::= conditions ("&&" conditions)*;
40
41 InstanceofCondition ::=  fragment "instanceof" type;
42
43 IsBoundCondition    ::=  "isbound" slot "on" fragment;
44
45 EqualsCondition     ::=  fragment1 "equals" fragment2;
46
47 ForEachLoop         ::=  "foreach" "(" fragmentName ":" list ")" "{" body "}";
48
49 Traverse            ::=  "traverse" "(" fragmentName ":" list ")" "{" body "}";
50
51 Prune               ::=  "prune" ";";
```

Listing B.7: Concrete syntax grammar of basic composition language

B.5 Composer Definition Language

```
1 ComposerDefinition = construct:componentmodel.AbstractFragmentType,
2                      arguments:compositionlanguage.AbstractFragmentName*,
3                      statementList:compositionlanguage.CompositionStatementList,
4                      returnFragment:compositionlanguage.AbstractFragmentName?;
```

Listing B.8: Abstract syntax grammar of composer definition language

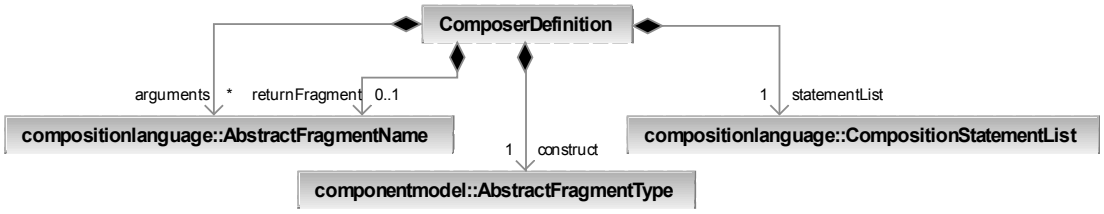

Figure B.5: Abstract syntax metamodel of composer definition language

```
1 CONCRETESYNTAX composer FOR composerdefinition EXTENDS fcp
2
3 ComposerDefinition    ::= "define" "composer" construct "(" arguments ("," arguments)*  ")"
4                           "{" statementList ("return" returnFragment ";")? "}";
```

Listing B.9: Concrete syntax grammar of composer definition language

B.6 Weaving Language

```
1  WeavingProgram = fragmentDeclarations:compositionlanguage.FragmentDeclaration*,
2                   weavings:Weave*, prints:compositionlanguage.Print*;
3
4  Weave      ==> compositionlanguage.Composer;
5
6  Weave      = type:componentmodel.FragmentType, core:compositionlanguage.FragmentName,
7               position:compositionlanguage.FragmentName, qualifier:Qualifier,
8               aspect:compositionlanguage.FragmentDefinition;
9
10 Qualifier  = Before | After;
11
12 Before;
13 After;
```

Listing B.10: Abstract syntax grammar of weaving language

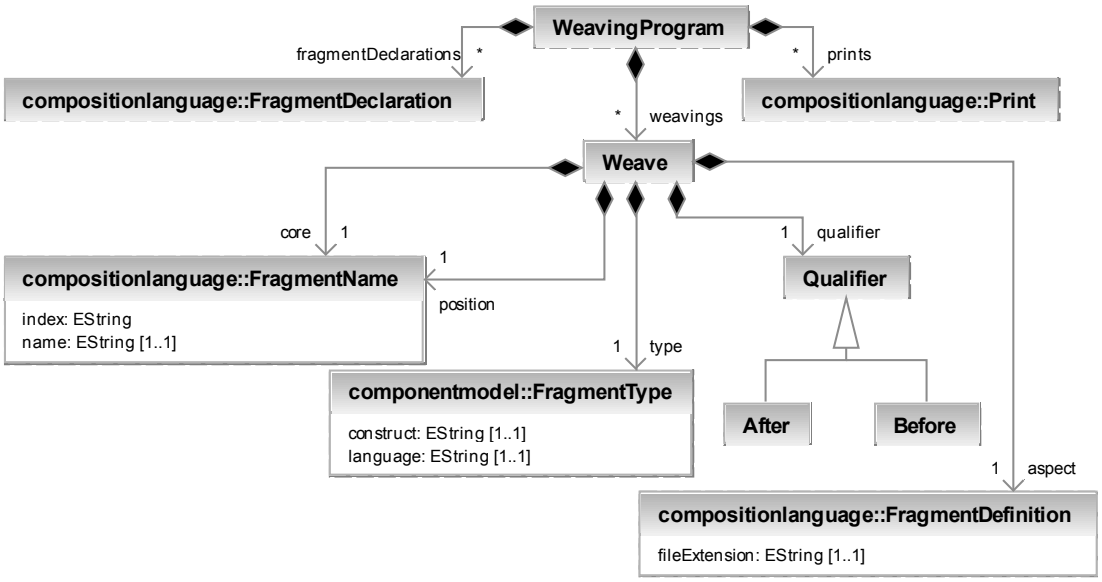

Figure B.6: Abstract syntax metamodel of weaving language

```
1  CONCRETESYNTAX weaving FOR weavinglanguage
2
3  WeavingProgram ::=  "cores" ":" (fragmentDeclarations ";")*
4                      "weavings" ":" weavings* "output" ":" (prints ";")*;
5
6  compositionlanguage.FragmentDeclaration ::=  "core" type name "=" value;
7
8  componentmodel.FragmentType ::=  (language[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*] ".")?
9                                   construct[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
10
11 compositionlanguage.FragmentName ::=  name[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*]
12                                       ("[" index[('-')? ('0'..'9')+] "]")? ("." subFragment)?;
13
14 compositionlanguage.Location ::=  path[('/' ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'.'|'-')+)+];
15
16 compositionlanguage.FragmentDefinition ::=  codePieces ("+" codePieces)* "."
17                                   fileExtension[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
18
19 compositionlanguage.ConcreteCodePiece ::=  code['\'' (~('\''))* '\''];
20
21 Weave     ::= "weave" type "in" core qualifier position "{" aspect "}";
22
23 Before    ::= "before";
24 After     ::= "after";
25
26 compositionlanguage.Print  ::=  "print" fragment "to" location;
```

Listing B.11: Concrete syntax grammar of weaving language

B.7 Java-

```
CompilationUnit      = packageDeclaration:PackageDeclaration?, importDeclarations:ImportDeclaration*,
                       classDeclarations:ClassDeclaration*;

PackageDeclaration   = name:QualifiedName;
ImportDeclaration    = name:QualifiedName;
ClassDeclaration     = modifier:Modifier, name:Identifier, extends:QualifiedName?,
                       implements:QualifiedName*, classBody:MemberDeclaration*;
MemberDeclaration    = AttributeDeclaration | MethodDeclaration;
AttributeDeclaration = modifier:Modifier, type:QualifiedName, name:Identifier, value:Value?;
MethodDeclaration    = modifier:Modifier, type:QualifiedName, name:Identifier,
                       arguments:VariableDeclaration*, statements:Statement*;

Statement            = MethodCall | VariableDeclaration | VariableAssignment;
VariableDeclaration  = type:QualifiedName, name:Identifier, value:Value?;
VariableAssignment   = name:Identifier, value:Value;
MethodCall           = method:QualifiedName, arguments:Value*, succeedingCall:MethodCall?;

Value                = StringValue | IntegerValue | Identifier | MethodCall;

Modifier             = PrivateModifier | PublicModifier | ProtectedModifier;

QualifiedName        = elements:Identifier+;

PrivateModifier;
PublicModifier;
ProtectedModifier;

StringValue   = value:S;
IntegerValue  = value:S;
Identifier    = value:S;
```

Listing B.12: Abstract syntax grammar of Java-

```
CONCRETESYNTAX java FOR java

CompilationUnit    ::= packageDeclaration? ";" (importDeclarations ";")* classDeclarations*;

PackageDeclaration ::= "package" name;
ImportDeclaration  ::= "import" name;
ClassDeclaration   ::=  modifier "class" name ("extends" extends)? ("implements" implements+)?
                        "{" classBody* "}";
AttributeDeclaration ::= modifier type name ("=" value)? ";";
MethodDeclaration  ::=  modifier type name "(" (arguments ("," arguments)*)? ")" "{" (statements ";")* "}";

VariableDeclaration ::= type name ("=" value)?;
VariableAssignment ::= name "=" value;
MethodCall         ::= method "(" (arguments ("," arguments)*)? ")" ("." succeedingCall)?;

QualifiedName      ::= elements ("." elements)*;

PrivateModifier    ::= "private";
PublicModifier     ::= "public";
ProtectedModifier  ::= "protected";

StringValue        ::= value['\"' (~('\"'|'\\') | '\\' . )*  '\"'];
IntegerValue       ::= value[('-')? ('0'..'9')+];
Identifier         ::= value[('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*];
```

Listing B.13: Concrete syntax grammar of Java-

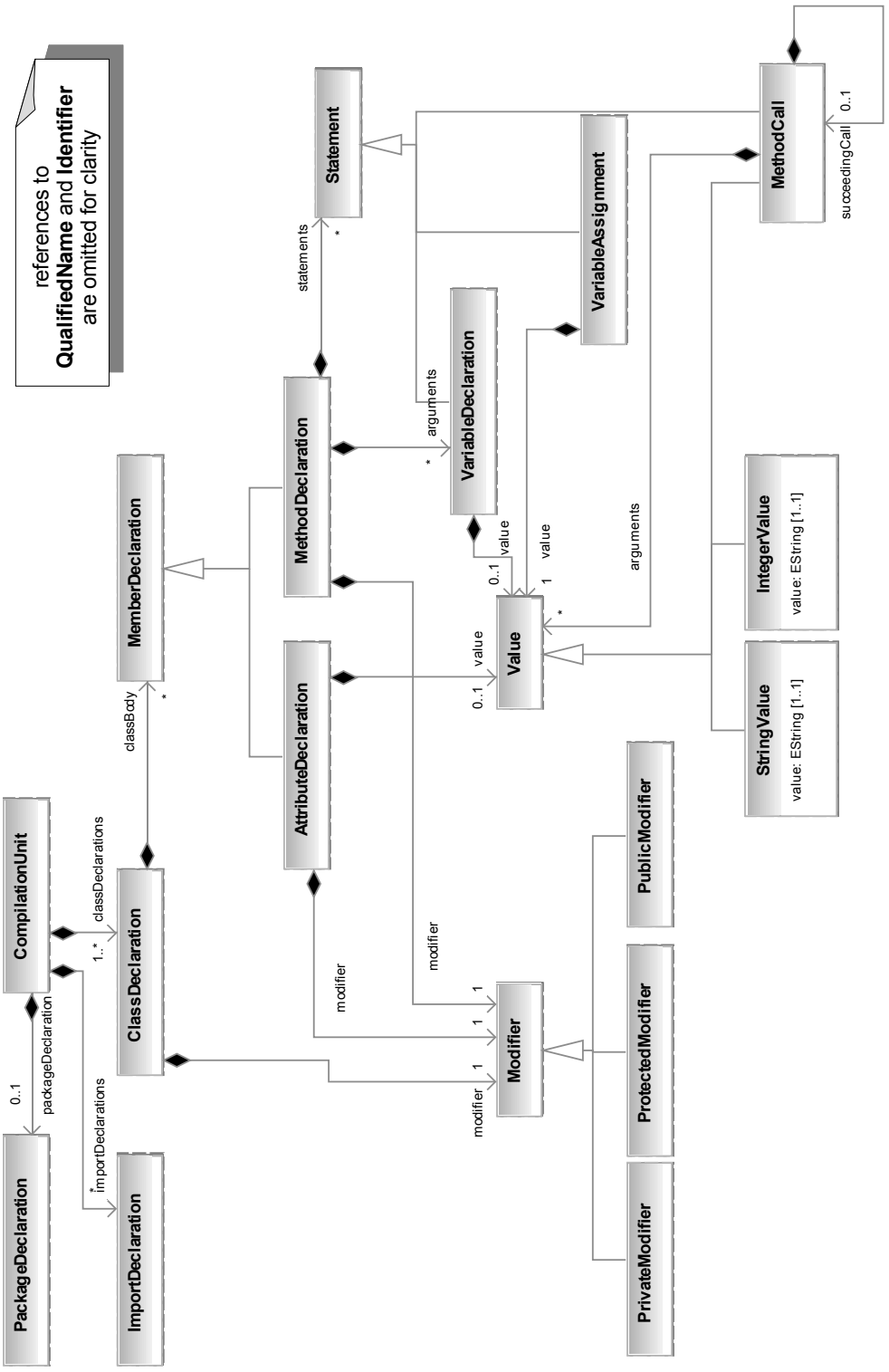

Figure B.7: Abstract syntax metamodel of Java-

B.8 Notation3

```
1  N3Doc          = statements:Statement+;
2
3  Statement      = Directive |  Triple;
4
5  Directive      = prefixName:Name?, uriref:Uriref;
6
7  Triple         = subject:Subject, predicateObjectList:AbstractPredicateObject+;
8
9  AbstractPredicateObject = PredicateObject;
10
11 PredicateObject = verb:Verb, objectList:Obj+;
12
13 Verb           = Predicate | IsA;
14
15 Subject        = Resource | Blank;
16
17 Predicate      = resource:Resource;
18
19 Obj            = Resource | Blank | Literal;
20
21 Literal        = DatatypeString | NumberLiteral | BooleanLiteral;
22 DatatypeString = type:QuotedString, resource:Resource;
23 NumberLiteral  = value:S;
24 BooleanLiteral = BTRUE | BFALSE;
25 BTRUE;
26 BFALSE;
27
28 Blank          = NodeID | EmptyList | PredicateObjectList | Collection;
29
30 EmptyList;
31 PredicateObjectList = predicateObjects:AbstractPredicateObject*;
32 ItemList       = obj:Obj+;
33 Collection     = itemList:ItemList?;
34
35 Resource       = Uriref | Qname;
36
37 NodeID         = name:Name;
38
39 Qname          = prefix:Name?, name:Name?;
40
41 IsA;
42 Uriref         = value:S;
43 Name           = value:S;
44 QuotedString   = value:S;
```

Listing B.14: Abstract syntax grammar of Notation3

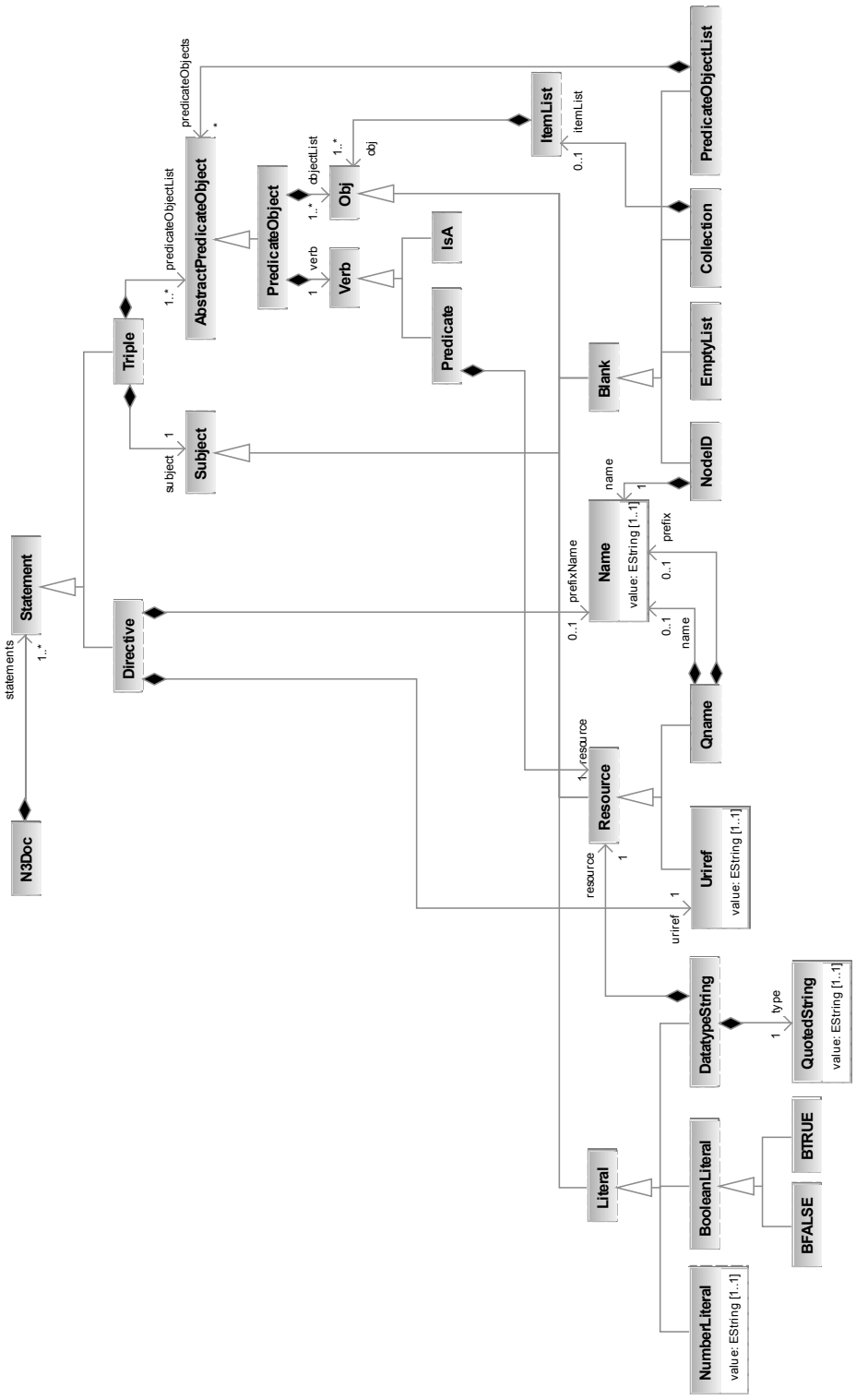

Figure B.8: Abstract syntax metamodel of Notation3

```
1  CONCRETESYNTAX n3 FOR n3
2
3  N3Doc       ::= (statements '.')+;
4
5  Directive   ::= '@prefix' prefixName? ':' uriref;
6
7  Triple      ::= subject predicateObjectList (';' predicateObjectList)* ';'?;
8
9  PredicateObject ::= verb objectList (',' objectList)*;
10
11 Predicate   ::= resource;
12
13 DatatypeString ::= type  '^^' resource;
14 NumberLiteral  ::= value[('+'|'-')? ('0'..'9')+ ('.' ('0'..'9')+ )?];
15 BTRUE       ::= 'true';
16 BFALSE      ::= 'false';
17
18
19 PredicateObjectList ::= '[' predicateObjects (';' predicateObjects)*  ']';
20 EmptyList   ::= '[' ']';
21 ItemList    ::= obj+;
22 Collection  ::= '(' itemList? ')';
23
24 NodeID      ::= '_:' name;
25
26 Qname       ::= prefix? ':' name?;
27
28 IsA         ::= 'a';
29 Uriref      ::= value['<' (~('>'|'<'))* '>'];
30 Name        ::= value[('A'..'Z' | '_' | 'a'..'z')('A'..'Z' | '_' | 'a'..'z' | '-' | '0'..'9')*];
31 QuotedString ::= value[('"' (~'"')* '"') | ('\'' (~'\'')* '\'')];
```

Listing B.15: Concrete syntax grammar of Notation3

B.9 ReuseNotation3

```
1  SubjectSlot;
2  n3.Subject       = SubjectSlot;
3  SubjectSlot    ==> componentmodel.Slot;
4
5  PredicateSlot;
6  n3.Predicate     = PredicateSlot;
7  PredicateSlot  ==> componentmodel.Slot;
8
9  ObjectSlot;
10 n3.Obj           = ObjectSlot;
11 ObjectSlot     ==> componentmodel.Slot;
12
13 PredicateObjectSlot;
14 n3.AbstractPredicateObject = PredicateSlot;
15 PredicateObjectSlot      ==> componentmodel.Slot;
```

Listing B.16: Abstract syntax grammar of ReuseNotation3

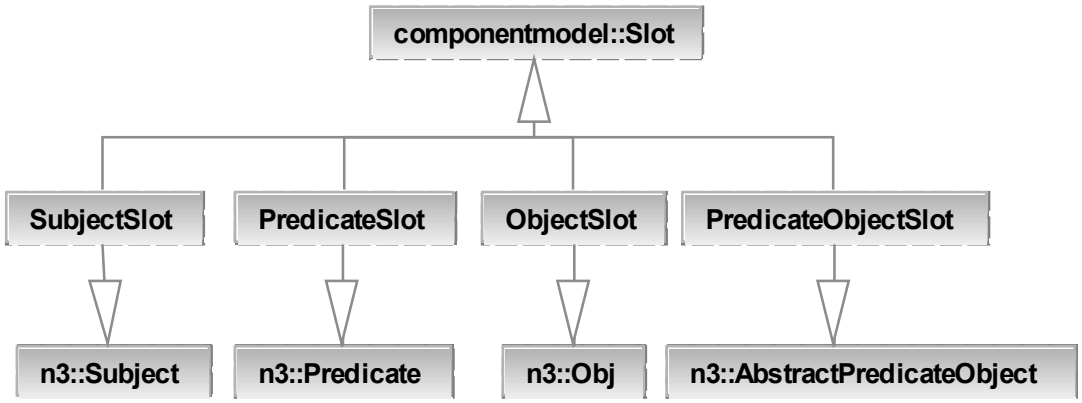

Figure B.9: Abstract syntax metamodel of ReuseNotation3

```
1 CONCRETESYNTAX rn3 FOR reusen3 EXTENDS n3
2
3 componentmodel.VariationPointName ::=
4                         name[('A'..'Z' | '_' | 'a'..'z')('A'..'Z' | '_' | 'a'..'z' | '-' | '0'..'9')*];
5
6 SubjectSlot          ::= "<<" name ">>";
7 PredicateSlot        ::= "<<" name ">>";
8 ObjectSlot           ::= "<<" name ">>";
9 PredicateObjectSlot ::= "<<" name ">>";
```

Listing B.17: Concrete syntax grammar of ReuseNotation3

B.10 N3 Pattern Composition Language

```
1 compositionlanguage.Composer = ClassAsPropertyValue1 | ClassAsPropertyValue2;
2
3 ClassAsPropertyValue1 = ontology:compositionlanguage.FragmentName,
4                         valueSlot:componentmodel.VariationPointName, valueClassName:n3.Resource;
5 ClassAsPropertyValue2 = ontology:compositionlanguage.FragmentName,
6                         valueSlot:componentmodel.VariationPointName, valueClassName:n3.Resource;
```

Listing B.18: Abstract syntax grammar of N3 Pattern Composition Language

```
1  CONCRETESYNTAX pc FOR n3pcl EXTENDS fcp
2
3  compositionlanguage.CompositionProgram  ::= "pattern" "composition" statementList;
4  n3.Qname   ::= prefix? ':' name?;
5  n3.Uriref  ::= value['<' (~('>'|'<'))* '>'];
6  n3.Name    ::= value[('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z'|'_'|'0'..'9')*];
7
8  ClassAsPropertyValue1 ::= "patternCAPV1" "(" "ontology" "=" ontology "," "valuepos" "=" valueSlot ","
9                            "class" "=" valueClassName ")" ";";
10 ClassAsPropertyValue2 ::= "patternCAPV2" "(" "ontology" "=" ontology "," "valuepos" "=" valueSlot ","
11                            "class" "=" valueClassName ")" ";";
```

Listing B.19: Concrete syntax grammar of N3 Pattern Composition Language

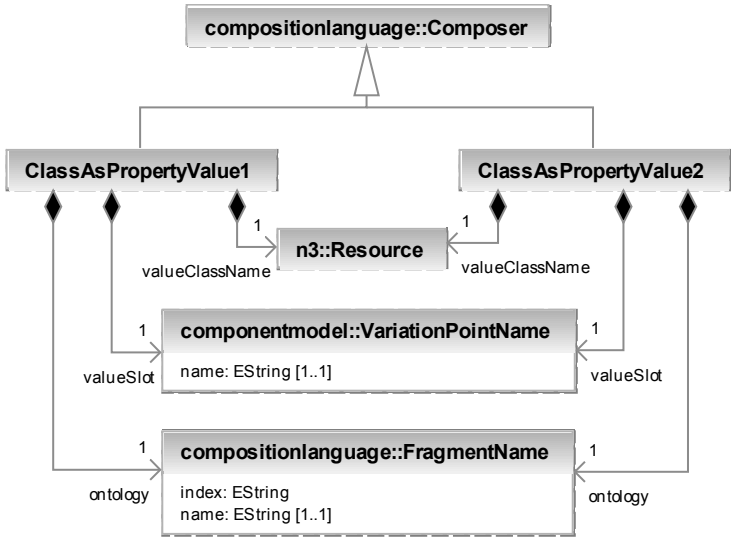

Figure B.10: Abstract syntax metamodel of N3 Pattern Composition Language

B.11 Xcerpt

```
Program            = statements:XcerptStatement+;
XcerptStatement    = GoalQueryRule | ConstructQueryRule;

ConstructQueryRule = construct:ConstructTerm, query:QueryTerm?;
GoalQueryRule      = goal:ConstructTerm,      query:QueryTerm?;

ConstructTerm      = ReferenceCt  | StructuredCt  | ContentCt  | VariableCt | ModifiedCt ;

ReferenceCt        = identifier:IdentifierCt ;
IdentifierCt       = NCName | IRI  | StringQuote | VariableCt;

ContentCt          = StringQuote | VariableCt ;

StructuredCt       = localSpec:LocalSpecCt,  children:ChildrenListCt;
ChildrenListCt     = OrderedChildrenListCt | UnorderedChildrenListCt;
OrderedChildrenListCt    = attributes:AttrTermListCt?, children:ConstructTerm*;
UnorderedChildrenListCt  = attributes:AttrTermListCt?, children:ConstructTerm*;
LocalSpecCt        = termIdentifier:TermIdentifierCt?, label:IdentifierCt;
TermIdentifierCt   = identifier:IdentifierCt;
AttrTermListCt     = attributes:AttrTermCt*;

AttrTermCt         = BaseAttrTermCt | VariableCt | ModifiedAttrTermCt;
BaseAttrTermCt     = label:IdentifierCt, content:ContentCt?;
VariableCt         =  name:NCName;

ModifiedCt         = GroupingCt  | OptionalCt;
ModifiedAttrTermCt = GroupingAttrTermCt  | OptionalAttrTermCt;
GroupingCt         = modifier:GroupingModifier, terms:ConstructTerm*,  grouped:Groupby?, ordered:Orderby?;
GroupingAttrTermCt = modifier:GroupingModifier, attribute:AttrTermCt?, grouped:Groupby?, ordered:Orderby?;
GroupingModifier   = All | Some | First;
All;
Some               = ammount:NumberCt;
First              = ammount:NumberCt;
```

```
Orderby            = variables:OptionalVariable*, relation:NCName?;
Groupby            = variables:OptionalVariable*, relation:NCName?;
OptionalVariable   = modifier:OptionalModifier?, variable:VariableCt;
NumberCt           = IntValue  | VariableCt ;

OptionalCt         = modifier:OptionalModifier, terms:ConstructTerm*, defaults:ConstructTerm*;
OptionalAttrTermCt = modifier:OptionalModifier, attributes:AttrTermCt*, defaults:AttrTermCt*;
OptionalModifier;

QueryTerm          = ModifiedQt | Resource | ReferenceQt  | ContentQt  | StructuredQt;
Resource           = resource:StringQuote, kind:StringQuote, queryTerm:QueryTerm?;
ModifiedQt         = VariableTermQt | LocationModifiedQt | OccurrenceModifiedQt | SelectionModifiedQt;

ReferenceQt        = identifier:IdentifierQt;
IdentifierQt       = NCName | IRI  | StringQuote | VariablenQt;
VariablenQt        = NamedVariableQt | AnonymousVariable;
NamedVariableQt    = name:NCName;
AnonymousVariable;

ContentQt          = StringQuote | VariablenQt;

StructuredQt       = localSpec:LocalSpecQt, children:ChildrenListQt;
ChildrenListQt     = OrderedChildrenListQt | UnorderedChildrenListQt |
                       OrderedPartialChildrenListQt | UnorderedPartialChildrenListQt;
OrderedChildrenListQt           = attributes:AttrTermListQt?, children:QueryTerm*;
UnorderedChildrenListQt         = attributes:AttrTermListQt?, children:QueryTerm*;
OrderedPartialChildrenListQt    = attributes:AttrTermListQt?, children:QueryTerm*;
UnorderedPartialChildrenListQt  = attributes:AttrTermListQt?, children:QueryTerm*;
LocalSpecQt        = termIdentifier:TermIdentifierQt?, label:IdentifierQt;
TermIdentifierQt   = identifier:IdentifierQt;

AttrTermListQt   = CompleteAttrTermListQt | PartialAttrTermListQt;
CompleteAttrTermListQt   = attributes:AttrTermQt*;
PartialAttrTermListQt    = attributes:AttrTermQt*;

AttrTermQt          = ModifiedAttrTermQt;
ModifiedAttrTermQt  = BaseAttrTermQt | VariableAttrTermQt | OccurrenceModifiedAttrTermQt |
                        SelectionModifiedAttrTermQt;
BaseAttrTermQt      = label:IdentifierQt, content:ContentQt?;
VariableTermQt      = variable:VariablenQt, baseTerm:QueryTerm;
VariableAttrTermQt  = variable:VariablenQt?, baseAttribute:BaseAttrTermQt;

SelectionModifiedQt          = modifier:SelectionModifier, terms:ModifiedQt*;
SelectionModifiedAttrTermQt  = modifier:SelectionModifier, attributes:ModifiedAttrTermQt*;
SelectionModifier;
OccurrenceModifiedQt         = modifier:OccurrenceModifier, terms:ModifiedQt*;
OccurrenceModifiedAttrTermQt = modifier:OccurrenceModifier, attributes:ModifiedAttrTermQt*;
OccurrenceModifier           = OptionalModifier  | WithoutModifier;
WithoutModifier;
LocationModifiedQt           = modifier:LocationModifier, terms:ModifiedQt*;
LocationModifier             = DescendantModifier  | PositionModifier;
DescendantModifier;
PositionModifier             = position:NumberQt;

NumberQt    = IntValue  | VariablenQt;

IRI         = value:S;
StringQuote = value:S;
IntValue    = value:S;
NCName      = value:S;
```

Listing B.20: Abstract syntax grammar of Xcerpt

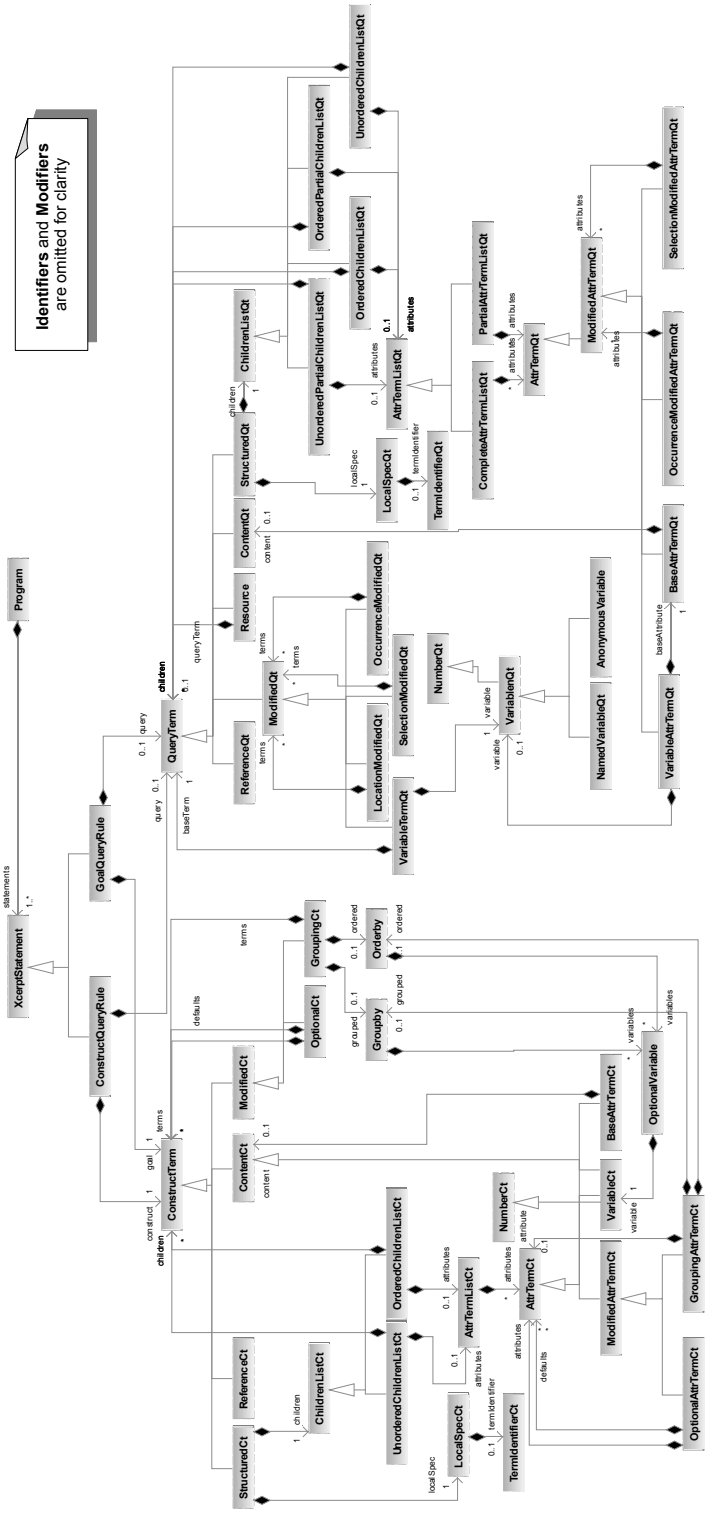

Figure B.11: Abstract syntax metamodel of Xcerpt

```
1  CONCRETESYNTAX xcerpt FOR xcerpt
2
3  Program            ::= statements+;
4
5  ConstructQueryRule ::= "CONSTRUCT" construct ("FROM" query)? "END";
6  GoalQueryRule      ::= "GOAL"      goal      ("FROM" query)? "END";
7
8  ReferenceCt        ::= "^" identifier;
9
10 StructuredCt       ::= localSpec children;
11 OrderedChildrenListCt   ::= "["( (attributes ",")? children ("," children)* | attributes? )"]";
12 UnorderedChildrenListCt ::= "{"( (attributes ",")? children ("," children)* | attributes? )"}";
13 LocalSpecCt        ::= termIdentifier? label;
14 TermIdentifierCt   ::=  identifier "@";
15 AttrTermListCt     ::=  "attributes" "{" (attributes ("," attributes)*)? "}";
16
17 BaseAttrTermCt     ::= label ("{" content "}")?;
18 VariableCt         ::= "var" name;
19
20 GroupingCt         ::= modifier (terms ("," terms)*)  grouped? ordered?;
21 GroupingAttrTermCt ::= modifier attribute? grouped? ordered?;
22 All                ::= "all";
23 Some               ::= "some" ammount;
24 First              ::= "first" ammount;
25 Orderby            ::= "group" "by" (variables ("," variables)*) relation?;
26 Groupby            ::= "order" "by" (variables ("," variables)*)  relation?;
27 OptionalVariable   ::= modifier? variable;
28
29 OptionalCt         ::= modifier  terms ("," terms)* ("with" "default" defaults ("," defaults)*)?;
30 OptionalAttrTermCt ::= modifier attributes ("," attributes)* ("with" "default" defaults ("," defaults)*)?;
31 OptionalModifier   ::= "optional";
32
33 Resource           ::= "in" "{" "resource" "{" resource "," kind "}" ("," queryTerm)? "}";
34
35 ReferenceQt        ::= "^" identifier;
36 NamedVariableQt    ::= "var" name;
37 AnonymousVariable  ::= "_";
38
39 StructuredQt       ::= localSpec children;
40 OrderedChildrenListQt           ::= "[" ( (attributes ",")? children ("," children)* | attributes? ) "]";
41 UnorderedChildrenListQt         ::= "{" ( (attributes ",")? children ("," children)* | attributes? ) "}";
42 OrderedPartialChildrenListQt    ::= "[["( (attributes ",")? children ("," children)* | attributes? )"]]";
43 UnorderedPartialChildrenListQt  ::= "{{"( (attributes ",")? children ("," children)* | attributes? )"}}";
44 LocalSpecQt       ::=  termIdentifier? label;
45 TermIdentifierQt  ::= identifier "@";
46
47 CompleteAttrTermListQt   ::= "attributes" "{"  (attributes ("," attributes)*)?  "}";
48 PartialAttrTermListQt    ::= "attributes" "{{" (attributes ("," attributes)*)? "}}";
49
50 BaseAttrTermQt      ::= label ("{" content "}")?;
51 VariableTermQt      ::= (variable "->") baseTerm;
52 VariableAttrTermQt  ::= (variable "->")? baseAttribute;
53
54 SelectionModifiedQt          ::= modifier terms ("," terms)*;
55 SelectionModifiedAttrTermQt  ::= modifier attributes ("," attributes)*;
56 SelectionModifier            ::= "except";
57 OccurrenceModifiedQt         ::= modifier terms ("," terms)*;
58 OccurrenceModifiedAttrTermQt ::= modifier attributes ("," attributes)*;
59 WithoutModifier              ::= "without";
60 LocationModifiedQt           ::= modifier terms ("," terms)*;
61 DescendantModifier           ::= "desc" ;
62 PositionModifier             ::= "pos" position;
```

```
63 IRI         ::= value['\"' (~('\"'|'\\') | '\\' . )*  '\"'];
64 StringQuote ::= value['\"' (~('\"'|'\\') | '\\' . )*  '\"'];
65 IntValue    ::= value[('0'..'9')+];
66 NCName      ::= value[('A'..'Z'|'a'..'z') ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')*];
```

Listing B.21: Concrete syntax grammar of Xcerpt (term syntax)

B.12 ReuseXcerpt

```
1  IdentifierSlot;
2  IdentifierSlot       ==> componentmodel.Slot;
3
4  xcerpt.IdentifierCt    = IdentifierSlot;
5  xcerpt.IdentifierQt    = IdentifierSlot;
6
7  CtSlot;
8  CtSlot               ==> componentmodel.Slot;
9  xcerpt.ConstructTerm   = CtSlot;
10
11 QtSlot;
12 QtSlot               ==> componentmodel.Slot;
13 xcerpt.QueryTerm       = QtSlot;
14
15
16 ImportComposer = moduleLocation:compositionlanguage.Location, ruleName:xcerpt.NCName, args:SlotValuePair*;
17 ImportComposer       ==> compositionlanguage.Composer;
18 xcerpt.XcerptStatement = ImportComposer;
19
20 SlotValuePair          = slot:componentmodel.VariationPointName, value:xcerpt.NCName;
21
22 xcerpt.XcerptStatement = ModifiedConstruct;
23 ModifiedConstruct      = modifier:ConstructModifier, construct:xcerpt.ConstructQueryRule;
24
25 ConstructModifier      = PublicModifier | PrivateModifier;
26 PublicModifier;
27 PrivateModifier;
```

Listing B.22: Abstract syntax grammar of ReuseXcerpt

```
1  CONCRETESYNTAX rxcerpt FOR reusexcerpt EXTENDS xcerpt
2
3  xcerpt.Program  ::= statements+;
4
5  IdentifierSlot  ::= "<<" name ">>";
6  CtSlot          ::= "<<" name ">>";
7  QtSlot          ::= "<<" name ">>";
8
9  componentmodel.VariationPointName ::= name[('A'..'Z'|'a'..'z') ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'-')*];
10 compositionlanguage.Location ::= path[('/' ('A'..'Z'|'a'..'z'|'0'..'9'|'_'|'.'|'-')+)+];
11
12 ImportComposer  ::= "IMPORT" moduleLocation "::" ruleName ( "[" "]" | "[" args ("," args)* "]") "END";
13 SlotValuePair   ::= slot "=" value;
14
15 ModifiedConstruct ::= modifier construct;
16 PublicModifier  ::= "PUBLIC";
17 PrivateModifier ::= "PRIVATE";
```

Listing B.23: Concrete syntax grammar of ReuseXcerpt

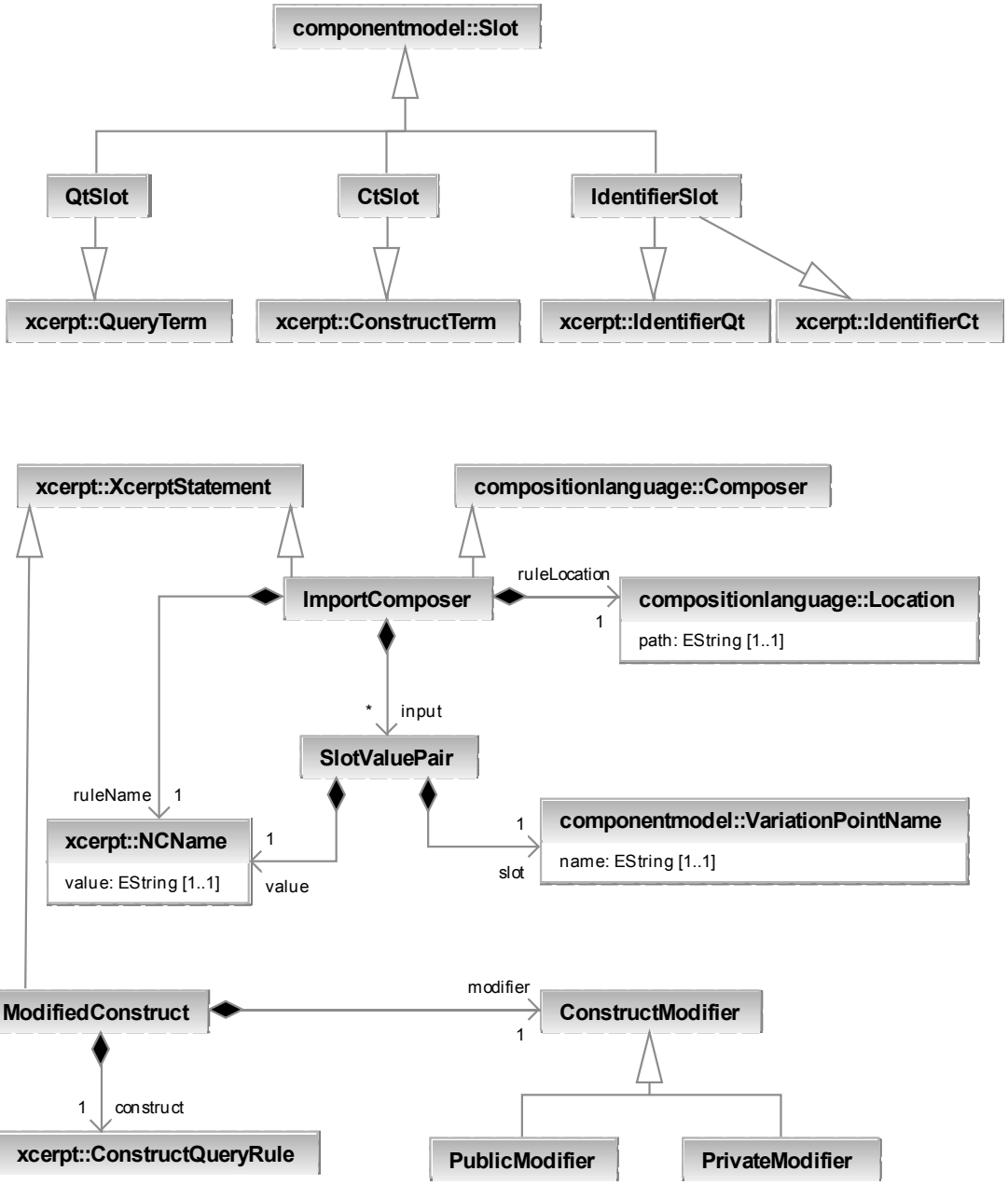

Figure B.12: Abstract syntax metamodel of ReuseXcerpt

C Glossary

ADL Architectural description language. A composition approach.

AOP Aspect-oriented programming. A composition approach.

basic composition language The language independent *composition language* of E-CoMoGen.

component language A language used to write components.

component model Component models define the structure of components and their *default* and *declared composition interface*.

composer See *composition operator*.

composition interface The interfaces of components that can be addressed in composition programs to compose them.

composition language Used to write *composition programs*.

composition operator Composition operators compose components. They implement a *composition technique*.

composition program Defines a system or a *program* by means of compositions.

composition system A composition system consists of a *component model*, a *composition technique*, and a *composition language*.

composition technique The way how components are composed.

construct See *language construct*.

core language A formal language that is extended with composition capabilities by means of a *reuse language*.

declared composition interface The explicitly declared part of a *composition interface*. In a *fragment component* it consists of *slots* and *declared hooks*.

declared hook Explicitly declared hook.

default composition interface The implicitly existing part of a *composition interface*. In a *fragment component* it consists of *implicit hooks*.

E-CoMoGen EMF based Component Model Generator. Language independent invasive composition tool to generate *invasive composition systems* and execute *composition programs*.

Eclipse Modeling Framework Java framework to construct, manage, and modify *models* and *metamodels*.

EMF See *Eclipse Modeling Framework*.

fragment See *fragment component*.

fragment component A source code fragment that is an *instance-of* a *language construct*. It is a valid component in an *invasive composition system* based on the corresponding *core language*. It may contain *slots* and *hooks*.

fragment type The *language construct* the fragment is an *instance-of*.

grammar A grammar describes the syntax of a formal language. It consists of grammar rules.

hook A point of extension in a *fragment component*.

HP Hyperspace programming. A composition approach.

implicit hook *Hook* that exists without being explicitly declared.

instance-of A *program* is an instance-of a *language construct* and of a *language description*. A *fragment component* is an instance-of a *language construct* and of a subset of a *language description*.

invasive composition system An invasive composition system consists of a *fragment component model*, a set of *composition operators*, and a *composition language*.

language construct Language constructs are parts of a language description. In *metamodels* they correspond to classes. In *grammars* they correspond to grammar rules. The terms *language construct* and *fragment type* are interchangeable.

language description See *grammar* and *metamodel*.

MBP Mixin-based programming. A composition approach.

metamodel In E-CoMoGen: a metamodel describes the *constructs* of a formal language (abstract syntax).

model In E-CoMoGen: a model corresponds to a *program*. It consists of *model elements*. It is an *instance-of* a metamodel.

model element In E-CoMoGen: a model element is a part of a *model*. It corresponds to a *fragment component*. It is an *instance-of* a part of a metamodel.

pattern composer *Composer* that implements a design or ontology pattern.

pattern composition Composition that employs *pattern composers*.

primitive composition operator Most simple *composition operator*. Every complex composition operator can be expressed as a combination of primitive ones.

program A complete compilable, interpretable, or executable unit written in some formal language.

reuse language The extension of syntax and semantics of the corresponding *core language* required for its superimposed composition capabilities.

slot A generic point in a *fragment component*.

VBP View-based programming. A composition approach.

Bibliography

[AJH+06] Uwe Aßmann, Jendrik Johannes, Jakob Henriksson, et al. E-CoMogen webpage. http://web.inf.tu-dresden.de/~jh30/work/rewerse/comogen, November 2006.

[AJHS06] Uwe Aßmann, Jendrik Johannes, Jakob Henriksson, and Ilie Savga. Composition of rule sets and ontologies. In *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 68–92. Springer, Heidelberg, 2006.

[asp05] The Aspect Blog @ aspectprogrammer.org. Aspect library discussion at AOSD 2005, March 2005. http://www.aspectprogrammer.org/blogs/adrian/2005/03/aspect_library.html.

[Aßm97] Uwe Aßmann. AOP with design patterns as meta-programming operators. Technical Report 28, Universität Karlsruhe, October 1997.

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of OOPSLA ECOOP '90*, number 25(10) in ACM SIGPLAN Notices, pages 303–311. ACM Press, New York, October 1990.

[Bec06] Dave Beckett. Turtle — terse RDF triple language. http://www.dajobe.org/2004/01/turtle, April 2006.

[BL06] Tim Berners-Lee. A readable language for data on the web: Notation 3, September 2006. http://www.w3.org/DesignIssues/Notation3.html.

[BLHC05] Tim Berners-Lee, Sandro Hawke, and Dan Connolly. Semantic Web tutorial using N3, May 2005. http://www.w3.org/DesignIssues/Notation3.html.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[BM04] Dave Beckett and Brian McBride. RDF/XML syntax specification, February 2004. W3C Recommendation. `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210`.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[Bür05] Torsten Bürger. Contributions to language composition using standard Semantic Web techniques. Master's thesis, Dresden University of Technology, October 2005.

[CC03] Xerox Corporation and Palo Alto Research Center. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`, 2003.

[Con06a] Gene Ontology Consortium. Gene ontology home. `http://www.geneontology.org`, September 2006.

[Con06b] The COMPOST Consortium. COMPOST webpage. `http://www.the-compost-system.org`, November 2006.

[F+04] Tim Furche et al. Identification of design principles, August 2004. REWERSE project report.

[F+06] Tim Furche et al. Initial draft of a language syntax, January 2006. REWERSE project report.

[Fou06] The Eclipse Foundation. Eclipse Modeling Framework — EMF. `http://www.eclipse.org/emf/`, November 2006.

[Gar03] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures, Third Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003), Advanced Lectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 1–24. Springer, September 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[Gro06a] Object Management Group. Meta Object Facility — MOF. `http://www.omg.org/mof/`, November 2006.

[Gro06b] Object Management Group. Unified Modeling Language — UML. `http://www.uml.org/`, November 2006.

[HOT02] William Harrison, Harold Ossher, and Peri Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM, 2002.

[Int96] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.

[Jav00] JavaSoft. *Enterprise Java Beans (TM)*, April 2000. Version 2.0.

[Joh06] Jendrik Johannes. Development of invasive composition systems for different languages based on ecore —the metamodel of the Eclipse Modeling Framework, June 2006. Großer Beleg, Dresden University of Technology.

[Kar06] Mattia Karlsson. Component based aspect weaving through invasive software composition. Master's thesis, Linköping University, August 2006.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax, February 2004. W3C Recommendation. `http://www.w3.org/TR/rdf-concepts`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Heidelberg, 1997.

[LMMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, Reading, MA, 1993.

[McI68] Doug McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[Mey90] Bertrand Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[NUW05] Natasha Noy, Michael Uschold, and Chris Welty. Representing classes as property values on the Semantic Web, April 2005. W3C Working Group Note. `http://www.w3.org/TR/swbp-classes-as-values`.

[OT00a] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 734–737, New York, NY, USA, 2000. ACM Press.

[OT00b] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Par06] Terence Parr. ANTLR — ANother Tool for Language Recognition — parser generator. `http://www.antlr.org`, November 2006.

[S+04] Ilie Savga et al. Report on the design of component model and composition technology for the datalog and prolog variants of the rewerse languages, August 2004. REWERSE project report.

[SB04] Sebastian Schaffert and Francois Bry. Querying the web reconsidered: A practical introduction to Xcerpt, April 2004.

[SB+06] Sebastian Schaffert, Francois Bry, et al. Xcerpt webpage. `http://www.xcerpt.org`, November 2006.

[Sie98] Jon Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Communications of the ACM*, 41(10):37–43, October 1998.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL web ontology language guide. Technical report, W3C Recommendation, February 2004. `http://www.w3.org/TR/owl-guide`.

[SWM06] Ralph Swick, Chris Welty, and Deborah McGuinness. Best practices and deployment working group: Ontology engineering and patterns task force (OEP), April 2006. `http://www.w3.org/2000/10/swap/doc/Overview.html`.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1998.

[Wüs05] Katja Wüst. Realisierung ausgewählter Entwurfsmuster als Compost-Operatoren, October 2005. Großer Beleg, Dresden University of Technology.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 28, 2006