Großer Beleg

Development Of Invasive Composition Systems For Different Languages Based On Ecore — The Metamodel Of The Eclipse Modeling Framework

submitted by

Jendrik Johannes

born 3-26-1981 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 June 8, 2006

Contents

List of Figures

1 Introduction

Many of today's languages used in software engineering have bad built-in support for software composition. It is important for software engineering to provide languages and tools to define, reuse and compose software components [1]. Still, many languages available today for programming, modeling, markup, scripting and rule writing do not support a proper way of defining components — a *component model.* They do not include concepts about how to compose components — a *composition language* — and thus there are no tools for these languages to execute compositions —a *composition technique.* A component model, a composition language and a composition technique make up a *composition system* [2].

Extending an existing language by a component model, to make it usable as a part of a composition system, could redeem the original language's weakness. Figure 1.1 shows a small program written in a simple programming language and one written in an extension of that language, which allows the definition of variation points.

```
program {
    x := 2;
    y := 5;

    z := 3 * x;
}
```

(a)

```
program {
    x := 2;
    y := 5;

    z := 3 * <<genericvar>>;
}
```

(b)

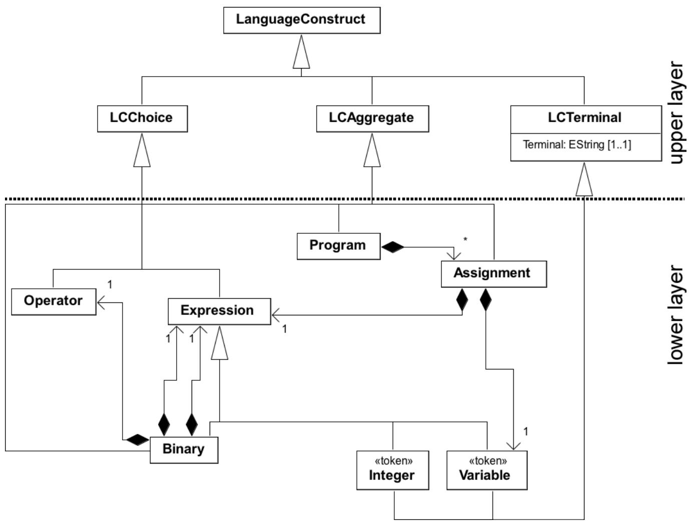
Figure 1.1: A program written in a simple programming language (a) and one written in an extension of that language (b). `<<genericvar>>` is a variation point where any variable name can be inserted when a composition is executed.

The Java and the XML language were successfully extended and deployed in a composition system (the COMPOST Framework [3]) by using the concepts and techniques of *Invasive Software Composition* (ISC) [2]. In the COMPOST Framework, the language extensions were implemented manually for the two supported languages. By formalizing the ISC concepts and techniques, a tool can be implemented which can extend any given language by a component model and deploy it in a composition system.

When working with different languages, it greatly facilitates if the languages are described in a common way. One popular way to describe the syntax of a language is EBNF [4]. A widely accepted approach in software engineering is to describe a system using models. UML [5] is the most wide-spread language today to define models. Languages are systems themselves and can thus be described by models. In fact, an EBNF grammar of a language is a model of that language. By describing the same language with the UML class diagram notation, one has another model of the language. Figure 1.2 shows two models of the same language.

```
program = "program",
    "{", {assignment}, "}";

assignment = variable,
    ":=", expression, ";";

expression = integer | binary |
    variable;

binary = "(", expression,
    operator, expression, ")";

operator = "+" | "-" | "*" | "/";

integer = ?( ('0'..'9')+ )?;

variable = ?( ('a'..'z')+ )?;
```

(a)



(b)

Figure 1.2: A small imperative programming language described by two different models: An EBNF grammar (a) and a UML class diagram (b). The program from Figure 1.1a is written in this language.

Different ideas about language modeling were presented in [6]. While it succeeded to model abstract syntax and static semantics of languages and gives ideas about extending these language models to integrate composition support, it does not go deeper into the concrete execution of composition.

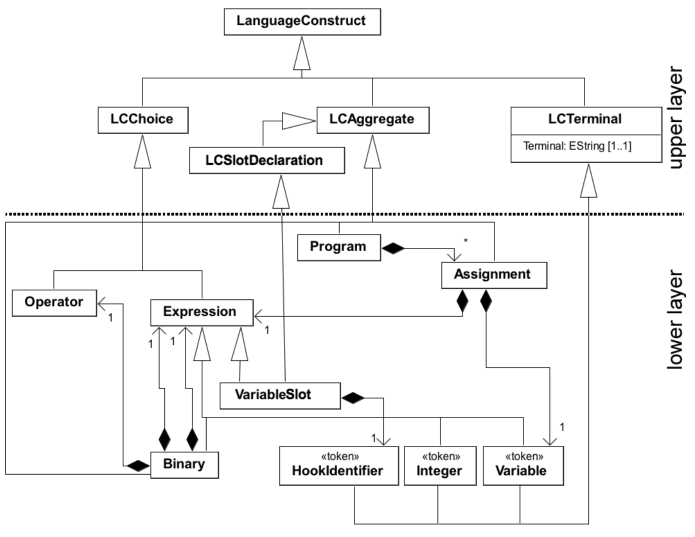
This thesis focuses around the extension of languages to support composition and its concrete application. The objective is to implement a tool which supports language extensions development as well as the processing and writing of programs and components in extended languages. To make an extended language usable, tools such as an integrated development environment (IDE) are needed. The popular *Eclipse Platform* [7], part of the open source *Eclipse Project* [8], is an excellent foundation for a customized IDE. Since it is desired to generate language tools automatically, another part of the Eclipse Project, the *Eclipse Modeling Framework* (EMF) [9], can also be deployed. It can generate code out of models based on the OMG's Meta Object Facility (MOF) modeling approach [10] and should therefore allow code-generation based on language models.

For MOF language modeling, the *two-layered language modeling* approach from [6] can be reused. It divides a language model into two layers such that all language constructs situated on the lower layer are based on concepts of the higher layer. All language models are based on the same higher layer, where concepts found in all languages are described. The model shown in Figure 1.2b contains such an upper layer of language concepts.

The concepts of invasive software composition can be introduced to the language models' upper layer. Any additional construct in an extended language will be based on an ISC concept (Figure 1.3). A tool which is aware of the common upper layer and knows how to resolve ISC based constructs in a generic way could execute the composition of programs written in any extended language. The only information such a tool would need is a model of the extended language that includes the common upper layer.

```
program = "program",
    "{", {assignment}, "}";

assignment = variable,
    ":=", expression, ";";

expression = integer | binary |
    variable | variable hook;

binary = "(", expression,
    operator, expression, ")";

operator = "+" | "-" | "*" | "/";

integer = ?( ('0'..'9')+ )?;

variable = ?( ('a'..'z')+ )?;

variable slot =
    "<<", slot identefier, ">>"

slot identifier = ?(('a'..'z')+)?
```

(a)



(b)

Figure 1.3: The Language extended with a new construct to support generic variable names. This new construct is based on the upper layer concept `LCSlotDeclaration`. The program from Figure 1.1b is written in this language.

1.1 The Topic

Building a tool for invasive software composition using the Eclipse Modeling Framework combined with a two-layered language modeling approach (as in [6]) looks promising. Therefore, this work will develop a two-layer language modeling approach based on MOF. It will describe the design of a generic ISC processing tool based on the Eclipse Modeling Framework and its code generation facilities. The result will integrate into the Eclipse Platform and be ready for further development into a full featured IDE to write programs and components in extended languages. The tool will be referenced to as *E-CoMoGen*[1] [11].

1.2 The Structure Of This Document

This thesis starts out with the initial ideas of a generic ISC tool, continues to describe the design of the tool and finally demonstrates the ready-to-use implementation with several examples. Chapter 2 introduces and explains modeling techniques used during development and runtime of the tool and also introduces the technologies which the tool is based on. The core functionality — processing of languages — and how it is modeled is described in Chapter 3. The overall structure of E-CoMoGen is documented in Chapter 4. Chapter 5 demonstrates its functionality by applying it to different formal languages.

[1]Eclipse Component Model Generator

2 Background

As stated in the introduction, this work is based on a modeling technique for languages — the two-layered language modeling approach introduced by [6] — and a technology — the Eclipse Modeling Framework [9]. To use the given modeling technique with the technology, this technique has to be applied to models the technology understands. Therefore, Section 2.1 examines how to define two-layered models in MOF-based modeling and discusses the meaning and relation of the terms *language*, *metalanguage*, *model* and *metamodel* in this context.

To extend languages and use them for composition, the concepts of invasive software composition, including *genericity* and *extensibility*, have to be introduced. That is done in Section 2.2.

Before looking at the design of the tool, one should have an understanding of the technology that is used. Therefore, an introduction into Eclipse and the Eclipse Modeling Framework is given in the Section 2.3.

2.1 Languages And Models

EBNF is a language in which context free grammars of other languages are written. Such languages are called *metalanguages*. Consider, for example, a program X, written in the language L, which grammar is written in the language EBNF. Then, EBNF is the metalanguage of L (Figure 2.1).

UML class diagrams are often referred to as *models* of a software system that is (going to be) implemented. The UML language itself is formally described by the *UML metamodel* which is modeled using the Meta Object Facility (MOF), which description is modeled by MOF itself. This leads to the OMG's four layer metamodel architecture visualized in Figure 2.2.

Often, the exact meanings of the words *model* and *metamodel* (and *meta-metamodel*) are not defined. In the OMG's definition, all seems clear at first. But is the UML metamodel always a metamodel? Or is that a question of point of view? EBNF is a language to describe L, which makes it also a metalanguage when recognizing that L is a language in itself. Maybe then the *UML metamodel* is a model of something, and a metamodel of something else? Maybe sometimes it is not a model at all? What exactly is a model and how do models relate to languages?

This section aims at giving clear answers to these questions, which are needed for the rest of this work. Therefore, the terminology of *metamodels* introduced by Favre [12, 13, 14] will be explained and used.

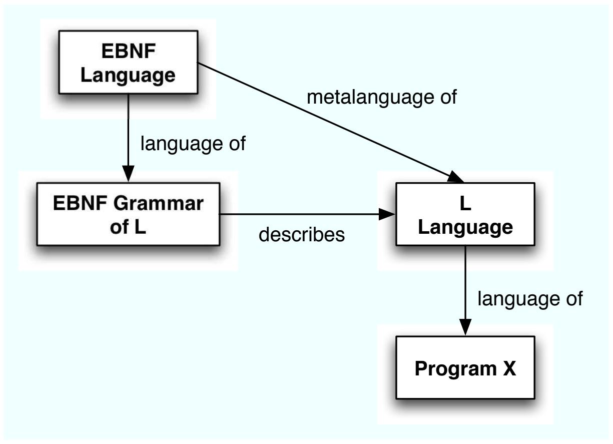


Figure 2.1: A metalanguage relationship.

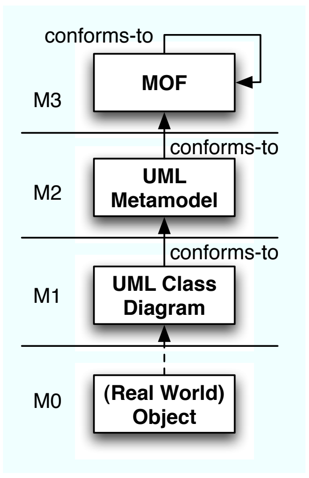


Figure 2.2: The MOF and UML in the OMG's four layer metamodel architecture [10].

2.1.1 Models

Favre defines the term model as a *role* a *system*[1] can play, implying that nothing is a model in general. A system plays the *role* of a *model* when it stands in relation to another system, which then plays the *role* of a *system under study* (SUS). The model gives answers about the SUS without the need to consider the SUS directly. The model is by no means required to give all answer about the SUS. Thus, in many cases, it is applicable to have more than one model for a SUS, which can give answers to different aspects of the SUS.

- The EBNF grammar of the language L is just an element of the EBNF language[2] in the first place. When seen in relation to the language L, the EBNF grammar of the language L is a model of the language L.
- A UML class diagram is a model of a real world system that the software is about to simulate, as well as it is a model of the implemented software.
- The UML metamodel is a model of the UML language.
- This text is a model of the E-CoMoGen tool since it describes many aspects of its functionality.

Note that the E-CoMoGen tool will have other models despite of this text. Tutorials and documentations on the website can also play the role of a model for E-CoMoGen.

[1]Systems are the primary elements in modeling. Everything is a system.
[2]When talking about a languages we refer to the (possible infinite) set of all valid sentences of the language.

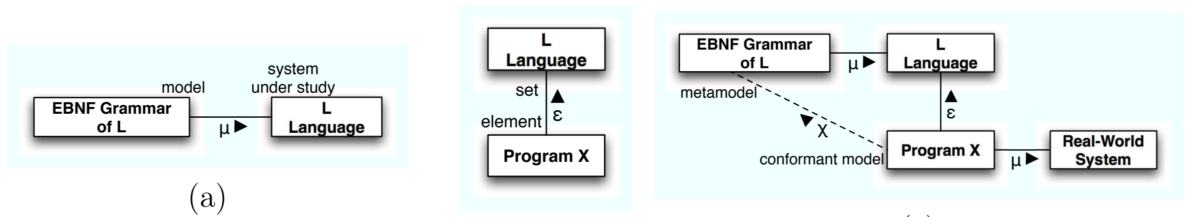
2.1.2 Metamodels And Megamodels

So where do metamodels come into play? To clarify relationships between systems (like the model-to-SUS relationship), Favre introduced the concept of *megamodels*. A megamodel consist of systems and relations between them. There are two relations already familiar to us. Here, Favre's notation is adopted.

- μ - **RepresentationOf, model / SUS**. Figure 2.3a. A model represents a system under study and gives answers about it.
- ε - **ElementOf, element / set**. Figure 2.3b. A program written in the Language L is an element of the L language (while the language is the, possible infinite, set of all possible L programs).

In some megamodels one can find *meta-step patterns*, which are characterized by the combination of two μ and one ε relation as illustrated in Figure 2.3c. This pattern leads to a new relation χ as a combination of μ and ε, describing the relation between a *model* and its *metamodel*. Note that the χ relation requires both μ relations to exist in order to identify both its ends as models.

- χ - **ConformsTo, conformant model / metamodel**.



(a) (b) (c)

Figure 2.3: The RepresentationOf (a), ElementOf (b) and ConformsTo (c) relationship. (c) as a whole is the appearance of a meta-step pattern.

Figure 2.4 shows a megamodel containing the languages previously discussed in this chapter. In the scenario there are two models of the language L. As in the introductory example, one is described in EBNF and one in UML. It is not further defined which language is used to describe EBNF itself. However, it could also be EBNF or UML.

The relation between the four layer metamodel architecture (Figure 2.2) and megamodels gets clarified by looking at meta-steps. There are two meta-steps in the metamodel architecture, which correspond to a layer change (there is no meta-step between the two lowest layers):

1. *possible runtime object states* **◀μ** *UML diagram* **χ▶** *UML metamodel*
2. *UML language* **◀μ** *UML metamodel* **χ▶** *MOF*

The fixed position is assumed that the runtime objects on the lowest layer are never models. In that case, all the UML diagrams on the next layer are models[3] (and never metamodels) and the *UML metamodel* on the layer above is always a metamodel for those models. In this understanding, the term *UML metamodel* makes sense.

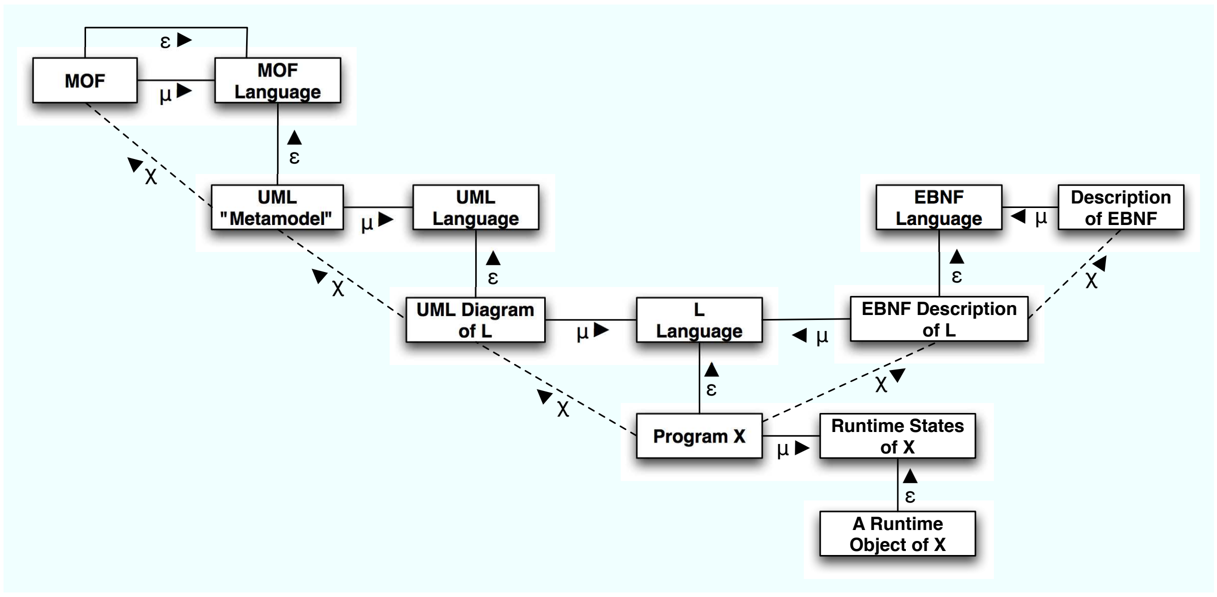


Figure 2.4: A megamodel of language modeling using EBNF and UML + MOF.

In the megamodel illustrated in Figure 2.4, there are three meta-steps in a row:

1. *runtime states* **◀μ** *program* **χ▶** *UML language diagram*
2. *L language* **◀μ** *UML language diagram* **χ▶** *UML metamodel*
3. *UML language* **◀μ** *UML metamodel* **χ▶** *MOF*

The reason for this is that the particular UML diagram here models a language. Therefore, the program, which is an element of that language, is situated one layer below the language diagram (while a diagram of the program would be situated at the same layer as the program itself).

[3]A UML diagram is commonly interpreted as a model (μ) of the code of an implementation but it can also be interpreted as a model (μ) of all possible runtime states, which a runtime object is an element of (ε). Then the argument holds that it is always a model and never a metamodel, because the runtime objects are never models (and the meta-step pattern does not occur).

During the design of an ordinary software system, such a detailed understanding of relations between models and metamodels is unnecessary (the runtime objects are never models). However, when designing software that should deal with software itself (where the runtime objects are models as well), the flexibility offered by megamodels helps to clarify the model, metamodel and other relationships between systems.

2.1.3 Two-Layered Models

Following the modeling approach from [6], a language model consists of two layers. The layers inside the model are independent of the layers in the metamodeling architecture, where the model as a whole keeps its position on one layer (Figure 2.5). Similar ideas are presented in [15] and [16].

To indicate the layering inside a model in a megamodel, the δ relation is used:

- δ - **DecomposedIn, composite / part**. A complex system is a composition of its parts.

A two-layered language model is the composition of its layers. The upper layer is part of every language model. A lower layer with individual language constructs can be found in every language model (Figure 2.6).

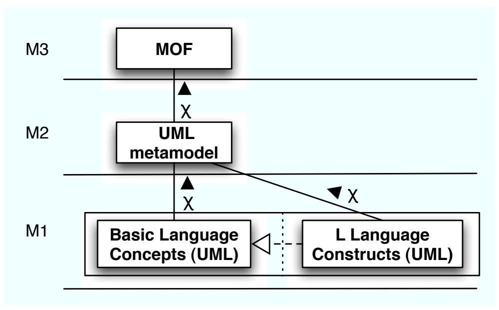


Figure 2.5: A language L modeled in UML. The model is divided into two layers.

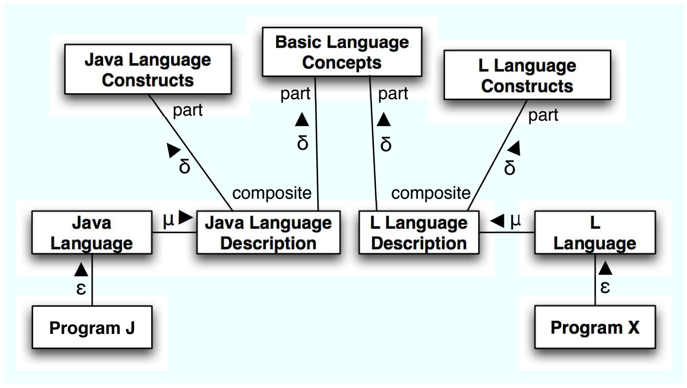


Figure 2.6: Language models as compositions of two layers.

2.2 Invasive Software Composition

A *composition system* consists of the following three major parts:

- A component model — What do components look like? What interface do they provide for composition?

- A composition technique — How are components connected (composed)?
- A composition language — A language which allows to write down composition programs, describing a composition of components.

Invasive Software Composition (ISC) [2, 17, 18] is a flexible and universal method for constructing software from components. *Fragments* are the reusable components in ISC. A fragment is an instance of a language construct[4] and is typed (the fragment's type is the language construct it is an instance of). Fragments may contain typed variation points — *slots* — and typed extension points — *hooks*, where other fragments of the correct type can be bound during composition. Slots and hooks make up the component model of a language used for ISC.

Composing fragments invasively means that internal parts of the fragments are merged during composition (Figure 2.7). This defines the composition technique. Usually, fragments consist of source code and the internal changes are done by rewriting code.

Composition programs, written in the composition language of an ISC system, include calls to *composers*, which execute the composition, and commands to load fragments into *fragment boxes* (Figure 2.7). In fragment boxes, declared slots and hooks can be addressed by the composers. Fragment boxes are typed corresponding to their content. Composers check, upon composition, if the type of a fragment box matches the type of the slot or hook it should be bound to. Thus, compositions are type-safe and errors can be reported at composition time.

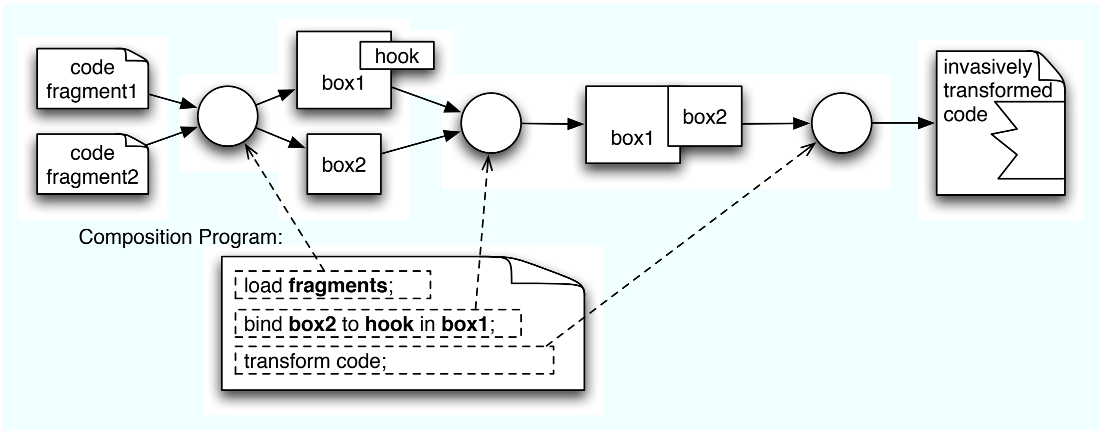


Figure 2.7: Two fragments are loaded into fragment boxes, which are composed by a bind composer. The result is invasively transformed code based on the original code fragments.

[4]An instance of a language construct (which can be defined by an EBNF rule or a class in a language model) is a piece of code, but not necessarily a compilable unit.

2.2.1 Universal Genericity And Extensibility

Universal genericity is supported by a language (as in BETA [19]), if it allows every piece of a fragment to be kept unspecified — *generic* — upon fragment definition [18]. These generic parts can later be filled by other fragments. This concept can be simulated in ISC by allowing a slot declarations for every possible language construct.

A language that has the possibility to extend every collection-like language construct[5] by additional fragments is called *universally extensible*. In ISC this concept is simulated when hooks are added to every instance of a collection-like language construct.

2.2.2 In-line Template Expansion

To ease component development, the composition technique can be enhanced by *in-line template expansion* [18]. In in-line template expansion, composers do not need to explicitly address a slot or hook to bind a fragment. Instead, a composer call acts as a slot itself. The fragment is then bound to the position where the composer call occured (Figure 2.8).

As a consequence, composer calls are allowed inside fragments, which requires a merge of the composition language and the language fragments are written in. This helps to write and to understand composition programs in the context of the composition as shown in Example 5.1.5.

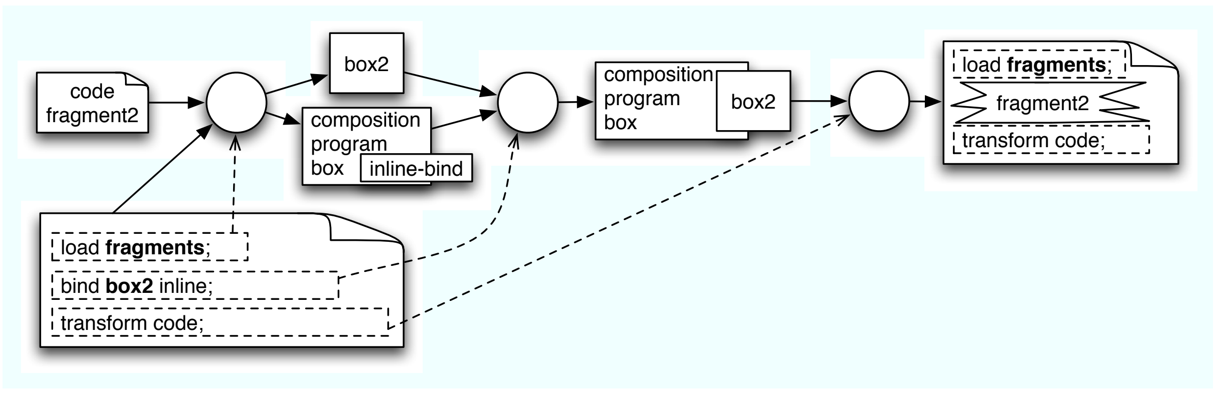


Figure 2.8: A fragment box is bound directly into a composition program.

[5]A language construct, which allows for an arbitrary long list of other language constructs (e.g. a list of variable declarations), is collection-like.

2.3 Eclipse And The Eclipse Modeling Framework

Eclipse [8] is an open source community focused on developing an open development platform (*Eclipse Platform* [7]) and application frameworks. The Eclipse Platform provides an highly flexible plugin mechanism which makes it easily extensible. This qualifies it as an excellent base for our implementation. The *Eclipse Modeling Framework* (EMF) [9] is one of the application frameworks developed by the Eclipse community. It defines its own metamodel — *Ecore* — for modeling, which design is highly influenced by the OMG's MOF specification, as it itself influenced the newest version of the specification [10].

In this chapter, the advantages gained by implementing an Eclipse based language modeling and processing system will be discussed. Furthermore, it explains how EMF can be used to generate code from MOF-based language models.

2.3.1 The Eclipse Platform

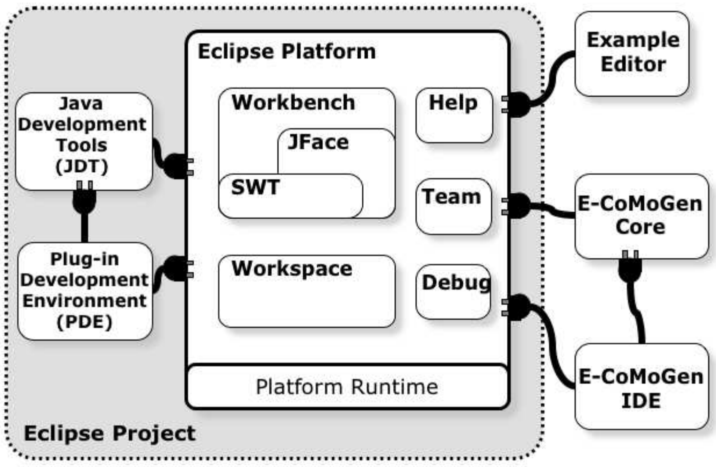


Figure 2.9: The Eclipse Platform: A composition of plugins. New plugins plug into the platform and into other plugins — a new editor as well as different parts of E-CoMoGen.

While the Eclipse Platform [7] can be used as the core of any tool or program, its full potential unfolds when it is used as the foundation of an IDE. The most prominent example is the *Eclipse JDT* (Java Development Tools) [20], which is one of the leading Java IDEs today.

Another Eclipse-based IDE is the *Eclipse PDE* (Plugin Development Environment) [21], which supports the developer in the process of writing plugins for the Eclipse Platform. The principles of the plugin concept are simple. The first thing one has to realize is that the whole Eclipse Platform is a composition of plugins (Figure 2.9). Every plugin defines a set of *extension-points* and provides *extensions* to extension-points defined by other plugins. With the definition of an extension-point, a contract is specified. Extensions have to fulfill the contract. In the simplest case, extensions are only required to provide a single string, but may also have to provide several Java classes implementing certain interfaces.

A simple example inspired by the plugin-with-editor template[6] stresses the eligibility of the Eclipse Platform as a base for a new IDE and demonstrates the application of *extensions* to *extension points*. The plugin `org.eclipse.ui` is part of the Eclipse Platform and provides the extension point `org.eclipse.ui.editors`. Writing an extension for this extension point requires to implement the interface `org.eclipse.ui.IEditorPart`. That implementation is the central class of the new editor. In its implementation, one can, for example, define rules for syntax highlighting. The extension can also be used to specify the files the editor is able to open. Later, when the plugin with the new extension is loaded into the Eclipse Platform, the new editor will be opened when an appropriated file is selected. Note that the whole process of file selection, providing a view on a file tree or opening the file and passing it to the editor, is managed by the Eclipse Platform. So are many other things that would need to be implemented when writing an editor from scratch. Editors are clearly just one part of an IDE. Functionalities like building, debugging and testing are other important functionalities which are supported by the Eclipse Platform.

It would be desirable to have an IDE in which components could be developed and composition programs could be written. The Eclipse Platform delivers the whole infrastructure of an IDE written in Java, which runs on all conventional operating systems. To create the E-CoMoGen IDE, extension to that infrastructure have to be implemented and plugged into the Eclipse Platform (Figure 2.9).

2.3.2 EMF And Ecore

As can be guessed from its name, EMF is a Framework which supports the creation of models. More precisely (to stick to the agreed terminology), it supports the creation of *systems* which are intended to play the role of a *model* for a system under study. These systems are called *core-models* in EMF. The metamodel for all core-models is called *Ecore*. Ecore is a core-model itself (it is defined in the Ecore language[7]) and is therefore its own metamodel. More precisely, a certain subset of the Ecore language (*Ecore kernel language*) is used to define a model (*Ecore*) of the the whole *Ecore language*. Figure 2.11 shows an UML class diagram of the Ecore kernel.

The UML metamodel and the MOF-model of the UML metamodel are both annotated as UML class diagrams. This can be interpreted as such: A subset of the UML language

[6]This template comes with the Eclipse Platform. Templates are sample implementations of extensions and can be used as guidelines when writing an extension.

[7]The Ecore language — the set of all possible core-models — is defined by a model (Figure 2.11).

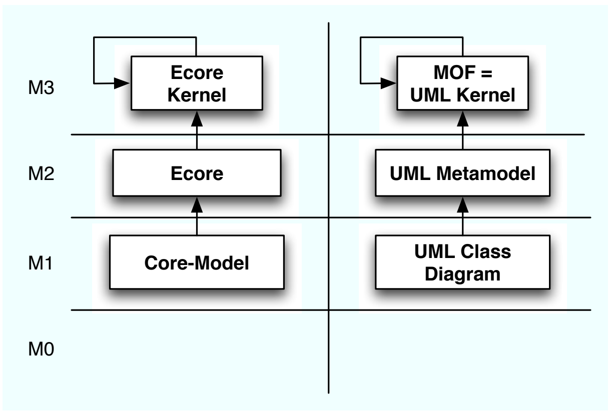


Figure 2.10: MOF and EMF modeling in the four layer metamodeling hierarchy with conforms-to relations (χ).

(*UML kernel language*) is used to define a model (*UML metamodel*) of the whole *UML language*. A model of the UML kernel language is then called *UML kernel*. Figure 2.10 shows the Ecore and MOF model-to-metamodel (χ) relations next to each other in the four layer metamodel architecture.

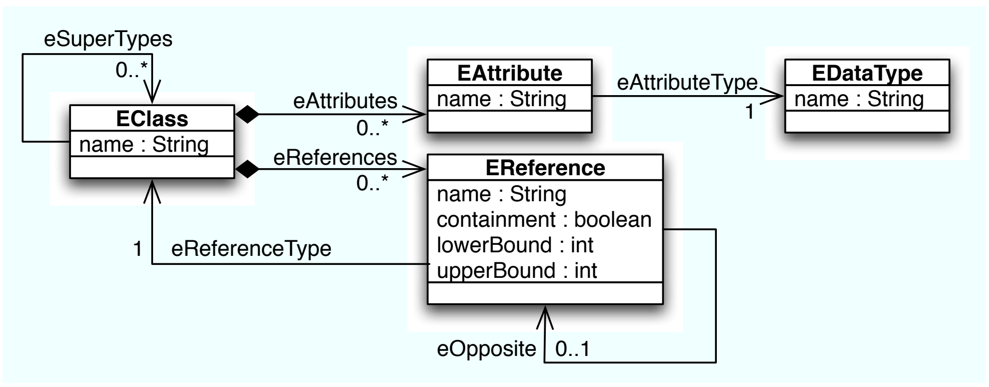


Figure 2.11: Ecore kernel — a model of the Ecore kernel language (Figure taken from [22]).

Even though the MOF specification is far more complex than that of Ecore, there are a lot of obvious similarities. Mapping MOF models to Ecore models is straightforward in many cases [23]. That statement is supported by the fact that UML class diagrams is a common annotation for core-models[8].

[8]There are different possibilities to define core-models. A mapping from UML class diagrams is one of them. A direct way to define and edit core-models is a graphical editor for the Eclipse Platform, which was employed for the implementation of E-CoMoGen.

Java code can be generated from core-models, resulting in a Java system which then plays the system-under-study role with respect to the core-model (Figure 2.12). The code generation is not examined here in detail. However, it is important to note that every class in the core-model will have a corresponding Java class implementation. It offers reflective methods by subclassing common EMF classes, which allows for examination and processing of runtime objects (conformant to a core-model) without knowledge of the concrete core-model. This is extremely helpful in an implementation which should be able to process language models unknown at implementation time.

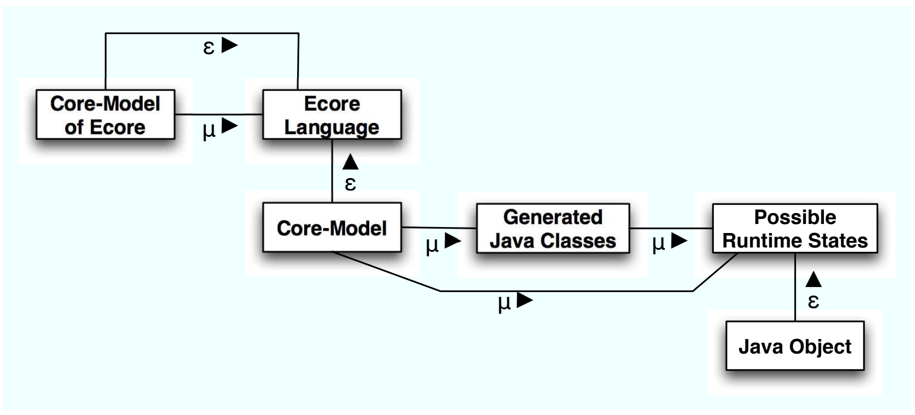


Figure 2.12: Relations of a core-model, generated Java code and runtime Java objects in a megamodel.

3 Language Modeling And Processing

This chapter describes how to create language models which can be processed by E-CoMoGen. A special emphasize is put on the modeling of invasive composition concepts and how E-CoMoGen can understand these concepts in order to execute the composition of components written in an extended language.

3.1 Describing Languages With Ecore

The previous chapter showed how languages can be modeled in a MOF modeling structure and also pointed out the similarities between MOF and Ecore. This knowledge is used in this section to derive a common structure for Ecore language models.

3.1.1 Modeling Common Language Concepts In Ecore

A core-model, called the *basic-language-concepts-core-model* (blc-core-model) (Figure 3.1), is defined, and it will be the upper layer of every language core-model. The classes in this model (the *concept-classes*) represent concepts common to all languages. Every *construct-class*[1] in a language core-model has to subclass one of these concept-classes and fulfill certain restrictions associated with it.

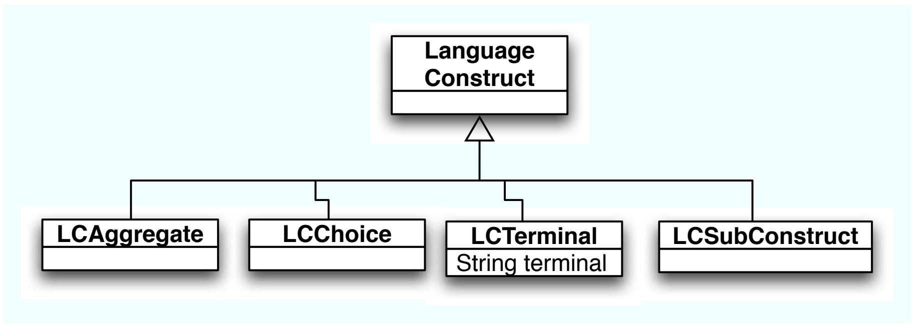


Figure 3.1: The basic-language-concepts-core-model.

The model is simple: The concept-classes merely have attributes or relations to each other. Nevertheless, the model is useful and necessary because of the restrictions associated to every concept-class. *LanguageConstruct* is the base class all other concept-classes

[1]A class representing a language construct.

inherit from. Below it are the three concept-classes *LCAggregate*, *LCChoice* and *LCTerminal*, as well as the additional concept-class *LCSubConstruct*. Every construct-class in a language core-model will subclass from these concept-classes. A construct-class can either be an aggregation of construct-classes (LCAggregate), a choice of construct-classes (LCChoice) or a representative of terminal symbols (LCTerminal). Additionally, aggregated construct-classes may contain sub-construct-classes (LCSubConstruct), which can not exist on their own.

The concepts behind the concept-classes are examined in detail in the next section. The restrictions they demand have to be taken into consideration when creating a language core-model. If the core-model is created by mapping another model (e.g. an EBNF grammar), the restriction can be enforced by the mapping.

The blc-core-model has many similarities to the EBNF language. This is practical since it should make it possible to describe a language in EBNF, and then map it to a core-model. Thus, a good understanding of the concepts in the blc-core-model can be gained by comparing it with the EBNF language.

3.1.2 Ecore And EBNF

The following list describes the main concepts behind EBNF and links each of them to the corresponding concept in the blc-core-model.

- **Language Constructs**: An EBNF grammar is a set of non-terminal rules that represent language constructs. A language core-model is a set of classes. For every rule in an EBNF description, there is a construct-class in a language core-model.

- **Identifier**: In EBNF, every rule is named by a so called metaidentifier. It corresponds to the name of the construct-class in the core-model.

- **Terminals**: A terminal is a concrete string sequence. It is annotated in EBNF by using one of the quote symbols (' or "). A construct-class which represents a terminal in a core-model is a subclass of *LCTerminal*. It has an annotation attached, which contains the terminal string or a regular expression representing a set of allowed terminal strings.

- **Aggregation**: To allow a certain sequence of non-terminals and terminals to occur in a program, a sequence can be defined on the right side of an EBNF rule. A construct-class in the core-model, which corresponds to a sequence rule, subclasses *LCAggregate*. It will have references to the construct-classes contained in the sequence. A reference might have three kinds of multiplicity — one, zero to one or any. That corresponds to the three possible kinds of sequences definable in EBNF — sequence, optional sequence and repeated sequence.

- **Choice**: Sometimes it is desired to define two or more rules as alternatives, which is possible in EBNF. Sublcasses of *LCChoice* represent such a choice in the core-model. Every construct, which is an option in a choice, subclasses the corresponding *choice-construct-class*. Thus, all the alternatives can take the position of the choice-construct-class.

- **Sub-Construct**: An EBNF rule can be a combination of different nested sequences, choices and terminals. Translating such a rule to a core-model requires it to be split up into more than one construct-class. This is the main difference between EBNF and language core-models. However, it does not lead to any problems. These additional classes are made into subclasses of *LCSubConstruct*, which has the benefit that they can be distinguished from explicitly defined constructs. This makes mapping from Ecore to EBNF possible both ways, without any deviations in the EBNF grammar even after multiple mappings back and forth.

In the introduction, an example of a small language was presented (Figure 1.2). Figure 3.2 shows the complete corresponding language core-model.

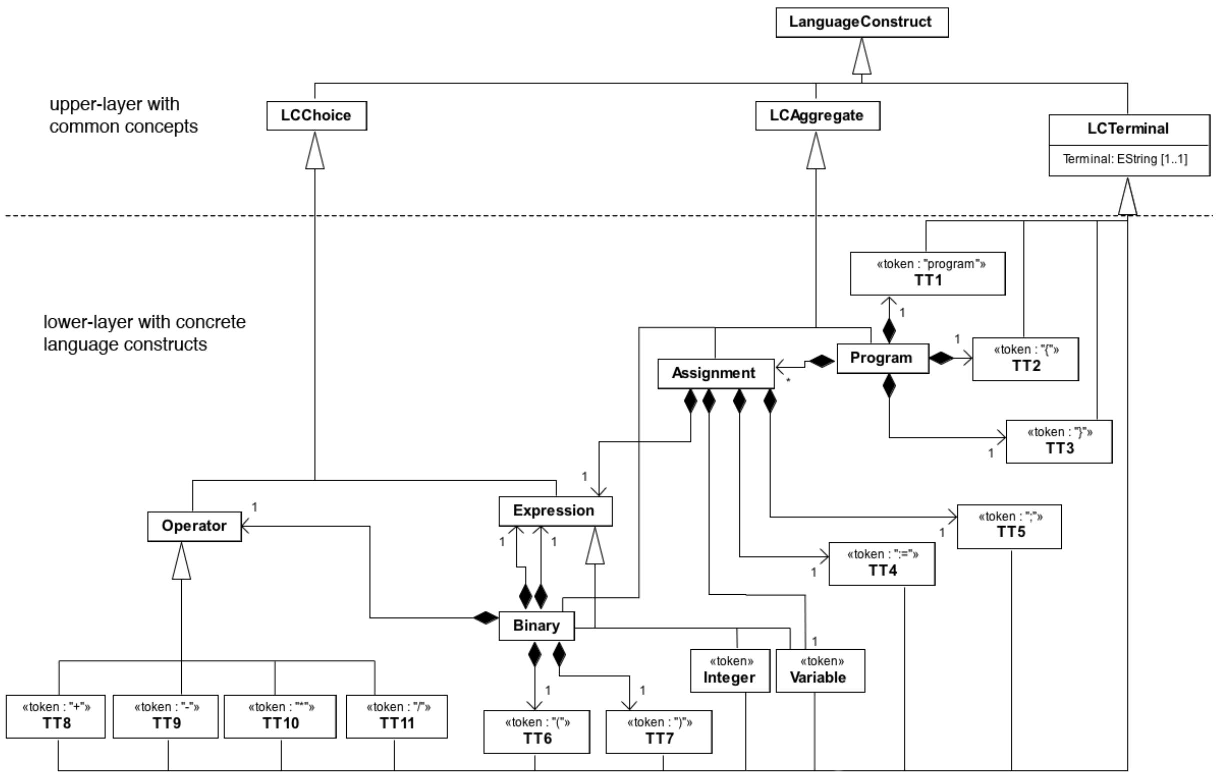


Figure 3.2: The complete core-model of the language from Figure 1.2

3.1.3 Abstract Vs. Concrete Syntax

The concept-classes in the blc-core-model differ slightly from the ones used in the upper layer for language models in [6]. While the differences will not be studied in detail here[2] their causes are explained in this section.

The work in [6] concentrated on formal and abstract descriptions of the syntax and semantics of languages. Therefore, no support for concrete syntax was integrated. For the E-CoMoGen tool, support of concrete syntax in language models is obviously necessary. The initial design of the blc-core-model was more strongly oriented towards the upper layer in [6]. Currently, however, it is stronger related to EBNF as previously described. A similar approach can be found in [24].

The question arises if a distinction between concrete and abstract syntax should be made, which is desirable in many cases of language processing [25]. The experiences made during the implementation showed that, for our purposes, a concrete-syntax-only based approach is applicable for many languages. This is because not just the input, but also the output, of the processing is written in concrete syntax. It might, however, not be appropriate for every language. Therefore, a distinction between concrete and abstract syntax might be desirable in future versions. That will on the other hand introduce new challenges to the mapping from and to EBNF.

3.2 Introducing ISC Concepts To Language Core-Models

If a language should be used for invasive software composition it has to be extended by a component model that allows for hook and slot declarations. Additionally, a composition language is needed to write composition programs that load fragments and call composers. From now on every given language will be called a *core language*. A language that is extended by new constructs to make it applicable in ISC is called a *reuse language*.

The *composition language* can be a part of the *reuse language* such that commands to load fragments into fragment boxes and to call composers can be embedded directly into fragments and programs. This has the advantage that the technique of *in-line template expansion* (Section 2.2.2) can be applied.

The following general ISC concepts for a reuse language that contains a composition language, are derived:

- **Slot/Hook Declaration**: *Slot* and *hook* declaration constructs are described by the component model of a reuse language. Every slot and every hook in a fragment has an unique identifier. A slot may be bound once, while a hook may be bound several times (extended), or not at all.

[2]Consult [6] to learn how the upper layer of language models looks there in detail.

- **Fragment Box Declaration**: A fragment box declaration is used to load a *fragment* into a *fragment box* for composition. Such a construct is part of the composition language. A unique identifier is given to the fragment box and the location to its content (the fragment) has to be provided (e.g. path to a file in the file system). Each fragment box is typed (see Section 2.2) and a fragment box declaration has to know the type of the fragment box it declares, since it is not necessarily revealed by the fragment alone[3].

- **Composer Call**: A composer call executes a *composer* that operates on fragment boxes, slots and hooks and is a part of the composition language. A composer call has to be linked to an implementation of a composer. It takes a set of arguments which usually contains fragment boxes that should be composed and slots and hooks that should be bound. There are two basic composers, *bind* and *extend*, which operate on slots and hooks respectively. They can be implemented in a generic way and work for every reuse language.

These new concepts are added to the blc-core-model (Figure 3.3). They all subclass *LCAggregate* because the language constructs based on these concepts will always have to be aggregations, as we will see in the next section.

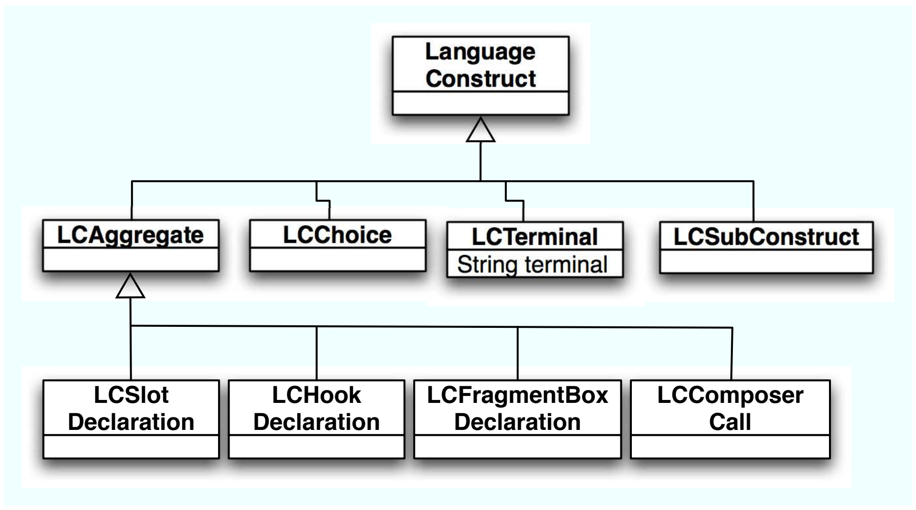


Figure 3.3: The basic-language-concepts-core-model enhanced with the ISC concepts.

[3]A fragment — a piece of code — can be interpreted as the instance of different language constructs which require equal concrete syntax.

3.2.1 Describing A Reuse Language With EBNF

When ISC constructs are added to a language, it is desirable to annotate this in the EBNF grammar since EBNF grammars for many languages do already exist. ISO-EBNF [4] defines the special sequence (`? {any character} ?`) as part of the EBNF language. This kind of sequence can be used for customized extensions to EBNF. Special sequences can be used to annotate ISC concepts and are interpreted as follows:

- **Named Sequences**: A special sequence that occurs as the first element of another sequence will be interpreted as the name of that sequence
- **Constant Values**: The content of a special sequence that occurs at any position, save the first, in another sequence, will be added to the syntax tree upon parsing as if it was written down in the program.
- **Regular Expressions**: An exception to the definitions above are sequences of the kind `?( {any character} )?`. These sequences are interpreted as regular expressions which can be used to define infinite sets of symbol sequences as a terminal symbol. E.g. all strings containing an arbitrary number of lower-case alphabetic characters can be expressed as `?( ('a'..'z')* )?`. The format of the regular expressions corresponds to the one used in ANTLR grammars [26].

Figure 3.4 illustrates the EBNF extensions with a small grammar and a program written in that grammar.

Some sequence names are defined to identify ISC concepts in the language description:

- **slotid or hookid**: The rest of a list named *slotid* or *hookid* makes up the identifier in a slot or hook declaration. The rule which contains this sequence is then recognized as a slot or hook declaration and the corresponding class in the core-model is a subclass of *LCSlotDeclaration* or *LCHookDeclaration*, respectively.
- **boxtype, boxid & boxlocation**: Lists of these three names have to appear in a rule to make it a fragment box declaration and the corresponding class in a core-model a subclass of *LCFragmentBoxDeclaration.* They identify the box's type, identifier and location, respectively.
- **composer**: An occurrence of a sequence named *composer* in a rule makes it an *LCComposerCall* subclass in the core-model and defines the composer's implementation. Other named sequences are likely to appear in the rule, based on which arguments the composer needs. The typical argument triple, expected by the bind and extend composers to bind a fragment box to a slot or a hook, is: *box*, *slot* or *hook* (the name of the slot or the hook in the box) and *value* (another box which should be bound to the slot/hook). To use the two composers for an in-line template expansion operation (Section 2.2.2), the *value* argument is sufficient.

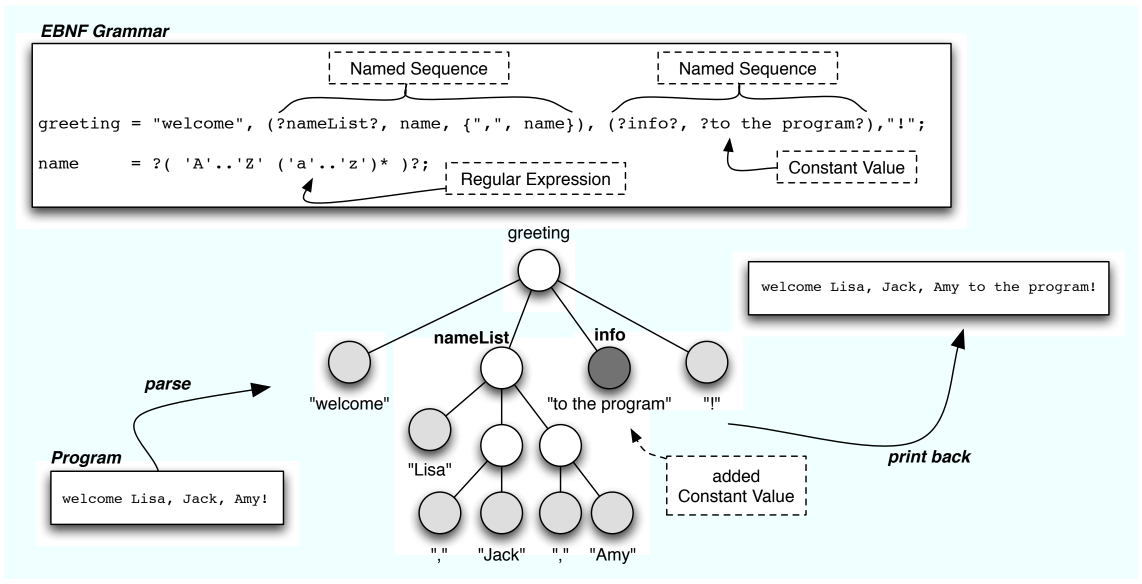


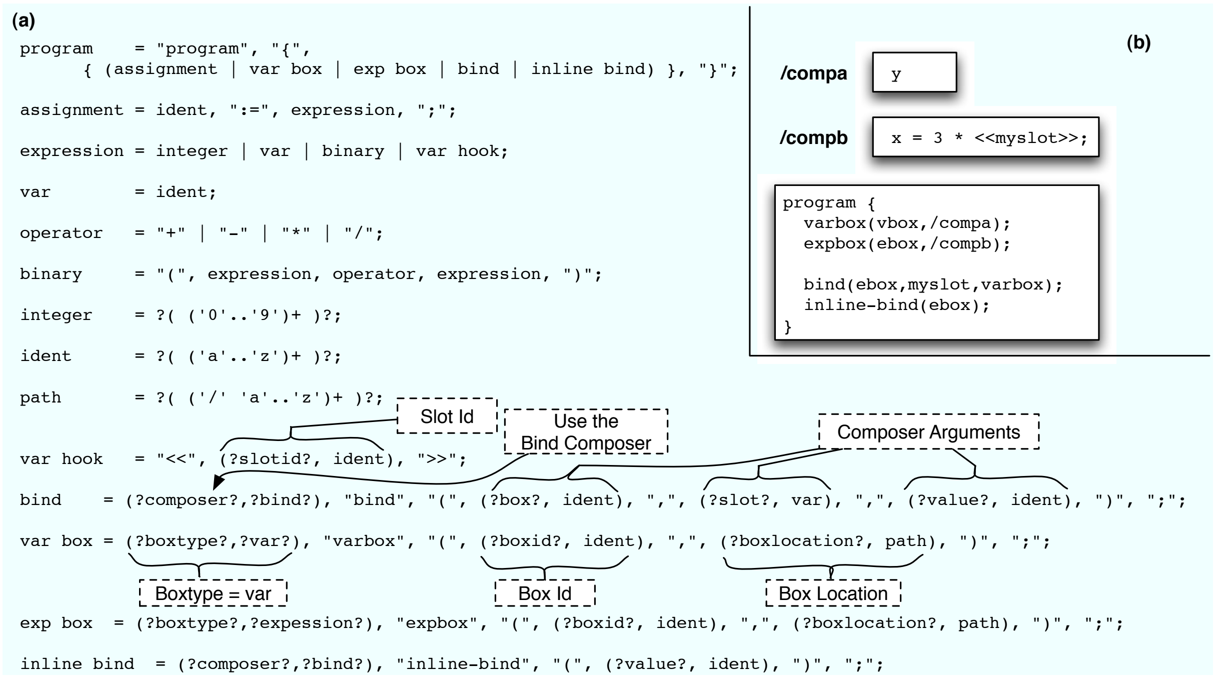
Figure 3.4: Special sequences in an EBNF grammar interpreted as named sequences, constant values and regular expressions.

The values for *composer* and *box type* are typically fixed. That is why constant values are needed. An extended example language illustrates the usage of the defined named sequences (Figure 3.5).

3.3 Composition By Tree Rewriting

With a reuse language core-model at hand, code written in that reuse language can be parsed into an instance of the core-model, which results in a tree of objects conforming to the core-model's construct-classes (the syntax tree). By examining the super classes of a construct-class, one can identify ISC concepts. In fact, the knowledge of the super classes is sufficient to perform composition. That is how the composition can be implemented without knowing the language it will actually work on.

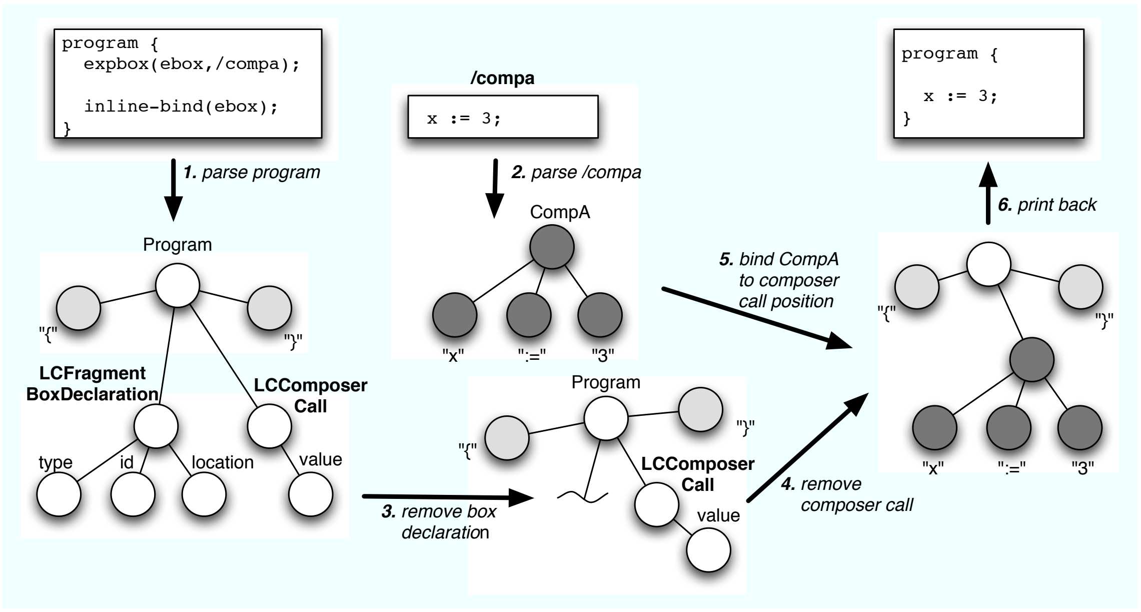
Composition is now basically reduced to tree rewriting. After the source code is parsed, the syntax tree is traversed and the ISC constructs are identified. If a fragment box declaration is encountered, its content is parsed into a tree as well. For composer calls, the corresponding composer implementation is determined and executed. The composer will then modify the syntax trees. The bind composer, for example, will take the trees of two components, find the slot in the first component and mount the tree of the second component to the slot's position. The extend composer works in a similar way on hooks. Loaded fragment box declarations and executed composer calls are removed from the tree (Figure 3.6).



```
program    = "program", "{",
      { (assignment | var box | exp box | bind | inline bind) }, "}";

assignment = ident, ":=", expression, ";";

expression = integer | var | binary | var hook;

var        = ident;

operator   = "+" | "-" | "*" | "/";

binary     = "(", expression, operator, expression, ")";

integer    = ?( ('0'..'9')+ )?;

ident      = ?( ('a'..'z')+ )?;

path       = ?( ('/' 'a'..'z')+ )?;

var hook   = "<<", (?slotid?, ident), ">>";

bind    = (?composer?,?bind?), "bind", "(", (?box?, ident), ",", (?slot?, var), ",", (?value?, ident), ")", ";";

var box = (?boxtype?,?var?), "varbox", "(", (?boxid?, ident), ",", (?boxlocation?, path), ")", ";";

exp box  = (?boxtype?,?expression?), "expbox", "(", (?boxid?, ident), ",", (?boxlocation?, path), ")", ";";

inline bind  = (?composer?,?bind?), "inline-bind", "(", (?value?, ident), ")", ";";
```

```
/compa   y

/compb   x = 3 * <<myslot>>;

program {
  varbox(vbox,/compa);
  expbox(ebox,/compb);

  bind(ebox,myslot,varbox);
  inline-bind(ebox);
}
```

Figure 3.5: An EBNF grammar with invasive composition constructs (a) and a program composing two components written in that language (b).



```
program {
  expbox(ebox,/compa);

  inline-bind(ebox);
}
```

```
/compa
x := 3;
```

```
program {

  x := 3;
}
```

Figure 3.6: A composition is executed by merging syntax trees.

With the extensibility of introducing new composers, the ISC tool can be used for many related composition techniques. Other more special composers could do complex rewriting and merging of syntax trees. Composer can for example be aspect weavers or connectors [2].

3.4 A Megamodel Of Language Models

To conclude this chapter, Figure 3.7 shows a megamodel of three languages: EBNF, Graal[4] and Reuse-Graal (a reuse language based on the core language Graal).

The interesting thing is that EBNF, Graal and Reuse-Graal can be handled equally, since they are all languages. For all of them, two models exists — an EBNF grammar and a core-model (the EBNF grammar of Reuse-Graal is omitted for clarity). These two models can be mapped to each other. Once the tool is up and running, the EBNF core-model (which is needed to describe another language) can be bootstrapped by using an EBNF grammar of EBNF (Appendix A).

Another thing worthy of remark is that the program written in Graal and the set of components written in Reuse-Graal are both models of the same runtime system, if the program is the result of the invasive composition of the components. Thus, invasive composition can be seen as model mapping as well.

[4]Graal is a simple imperative programing language which is used in [25] for teaching the theory of programing languages and which will be used in the first example in Chapter 5.

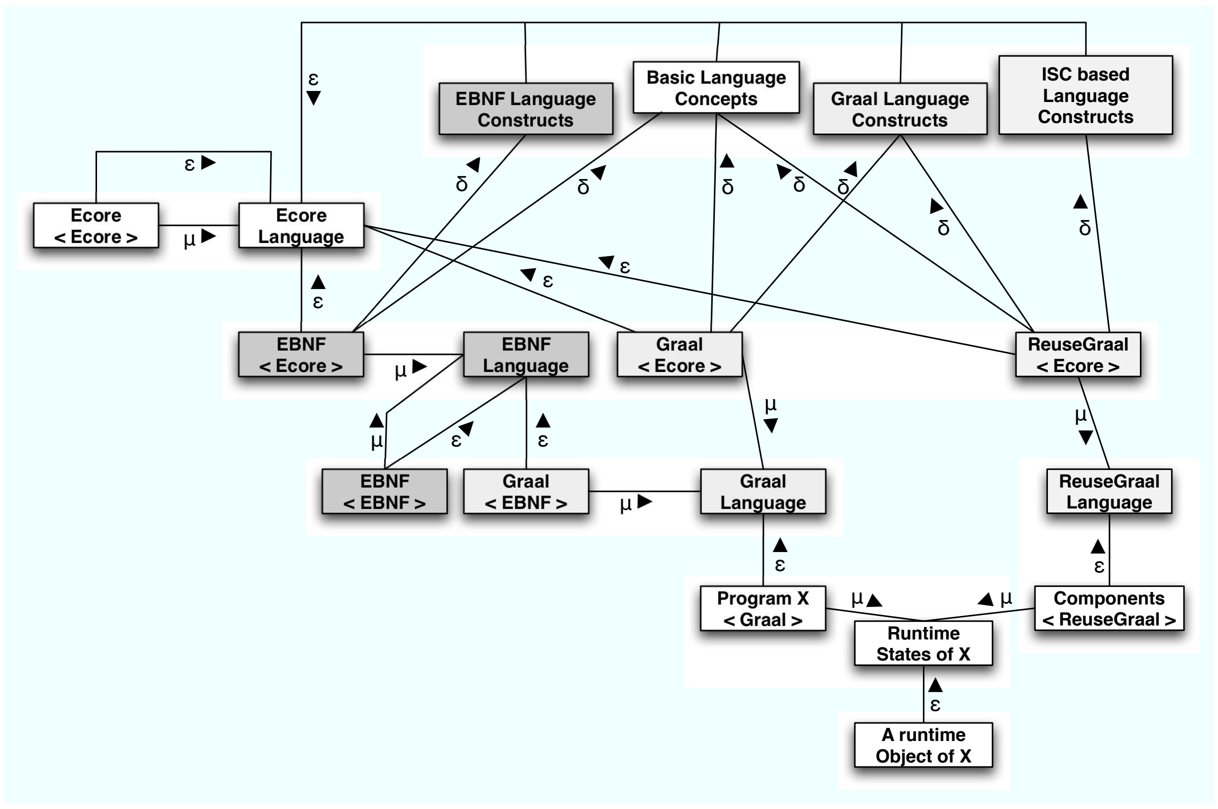


Figure 3.7: A megamodel of languages and their relations in Ecore language modeling.

4 Tool Implementation

The functionality of E-CoMoGen is twofold. The tool should be able to process the composition of fragments written in any reuse language, which can be parsed into an instance of a language core-model. Also, it should be able to produce such a core-model and the corresponding parser, when given an EBNF grammar of the reuse language. This chapter first describes the overall structure of the tool. It further includes one section on each of the tool's functionalities. Finally, it outlines the integration with the Eclipse Platform.

4.1 Tool Structure

The E-CoMoGen tool is made up of a core that consists of a set of Java classes, assembled in the *ecomogen.jar* file, and a collection of eclipse plugins (Figure 4.1). The core provides the main functionality of composing fragments. It also defines the interfaces for extending the tool. The code-generation plugin builds upon the EMF code-generation facilities to generate Java classes from language core-models and upon the ANTLR parser generator [26] to generate parsers for programs written in those languages. An IDE plugin integrates with the Eclipse Platform to execute compositions automatically and to give feedback of parsing and composition errors. The visualization and user interface parts are placed in separate plugins.

Languages, language mappings and composers are three possibilities of extending E-CoMoGen. For each of them some standard extension is implemented. A language plugin for the EBNF language provides classes generated from a language core-model of the EBNF language. Another plugin implements a mapping from EBNF to Ecore and back. With these plugins an EBNF description of any language can be translated into a core-model of that language from which a new language plugin can be generated by the code generation plugin. Note that the EBNF language plugin can be re-generated form an EBNF grammar of EBNF (Appendix A). Finally, the two fundamental composers of invasive composition — *bind* and *extend* — are provided in a generic implementation, working on any language core-model.

4.2 The E-CoMoGen Foundation

The central part of the tool includes the functionality for composing composition programs and components. The structure of its implementation will be explained in this section.

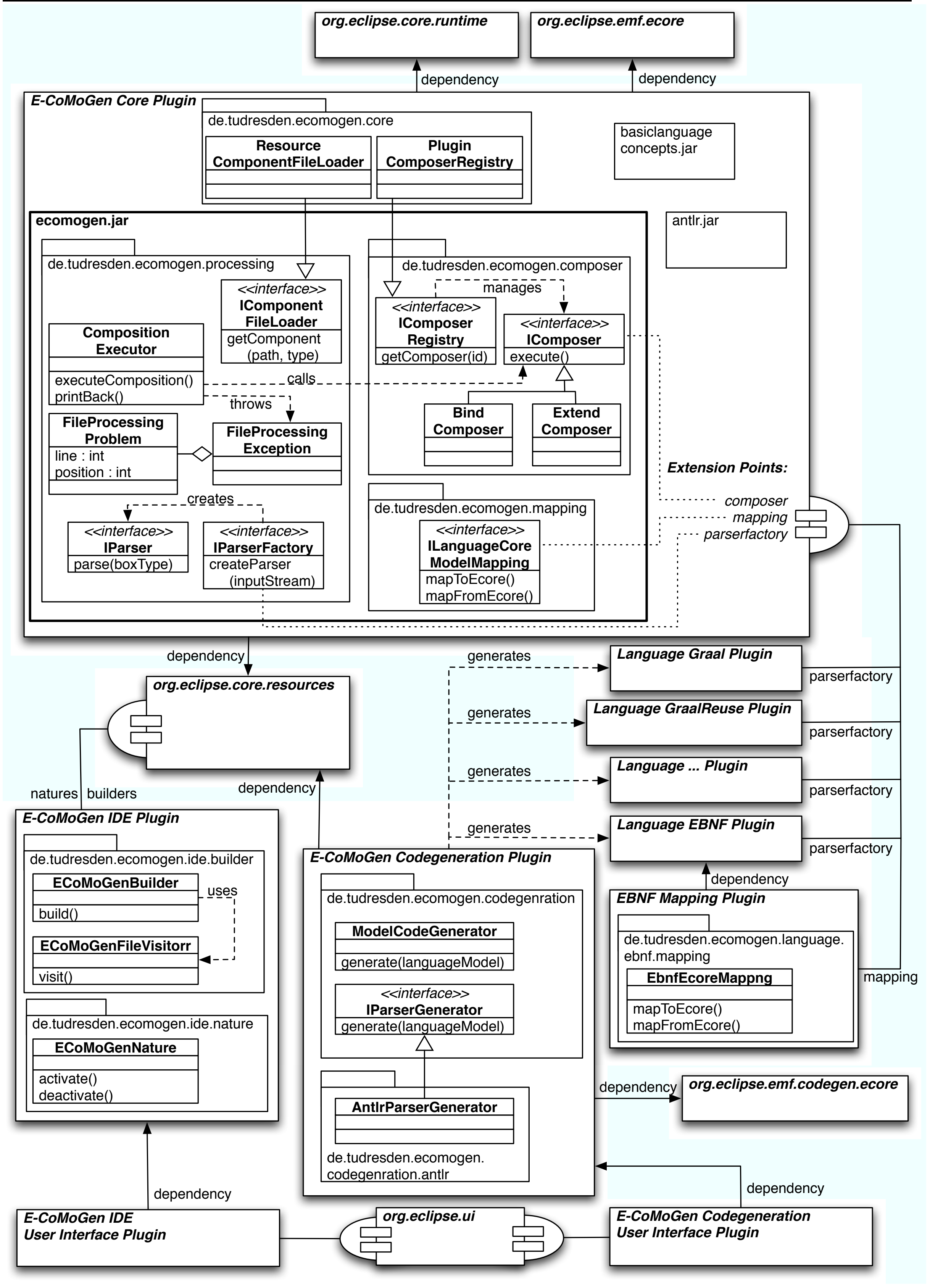


Figure 4.1: The structure of the E-CoMoGen tool. All E-CoMoGen plugins have dependencies to the Core Plugin which are not sketched.

4.2.1 The Central Packages In Ecomogen.jar

The heart of E-CoMoGen is assembled in the ecomogen.jar file. It can be used independently of the Eclipse Platform but has the following dependencies:

- **basiclanguageconcepts.jar**. This jar file contains classes generated from the basic-language-concepts-core-model. Classes generated from other language core-models depend on these classes.
- **emf.common.jar & emf.ecore.jar**. These jars are provided by the Eclipse Modeling Framework and are needed for working with core-models. They themselves do not have any other dependencies and can be used independent of other parts of EMF or Eclipse.
- **antlr.jar**. Parsers generated by ANTLR use common base classes. These classes are contained in this jar file available from [26].

The ecomogen.jar file contains the following packages and classes:

- **de.tudresden.inf.ecomogen.processing**. The processing package contains everything needed to process syntax trees of reuse language programs and components. It provides the interfaces *IParserFactory* and *IParser*, which language plugins will implement when they are generated (see Section 4.3). A parser parses fragment components to syntax trees which are then passed to the *CompositionExecutor*. It implements the syntax tree traversal (see Section 3.3). When syntax tree rewriting is finished, it can print back the result, by extracting all terminal symbols in the correct order from the composed syntax tree.

  During tree traversal, component files need to be loaded. Their location is identified by the *boxlocation* string in a composer-call (see Section 3.2.1). Since the structure of the underlying file system might differ (especially in non-Eclipse environments), the interpretation of the location string and the resulting file-loading can be varied by exchanging the implementation of *IComponentFileLoader*.

  Error reporting is supported by the classes *FileProcessingException* and *FileProcessingProblem*. If errors are encountered during parsing or composition, a *FileProcessingException* will be thrown with a set of *FileProcessingProblem*s attached. Each of these problems points at the position in the file where the problem occurred, which can be given as feedback to the user.

- **de.tudresden.inf.ecomogen.composer**. During tree traversal, composers are called. They are identified by a composer identifier (see Section 3.2.1). The corresponding implementation can be retrieved by querying an *IComposerRegistry*. All composers have to implement the *execute()* method of *IComposer*.

  The standard composer implementations, *BindComposer* and *ExtendComposer*, complete this package.

- **de.tudresden.inf.ecomogen.mapping**. To use EBNF as a metalanguage, a mapping from EBNF grammars to core-models needs to be provided (see Section 3.1.1). It is also desirable to employ other metalanguages, if required. *ILanguageCoreModelMapping* is the interface which needs to be implemented to map between language core-models and any other metalanguages[1].

4.2.2 The E-CoMoGen Core Plugin

When used in an Eclipse environment, the ecomogen.jar and its dependencies are wrapped by the E-CoMoGen core plugin, which provides additions for smooth integration into the Eclipse Platform. The plugin defines extension points for the three possible tool extensions (languages, mappings and composers) to make extensions easy and conformant with the usual way Eclipse plugins are extended. The *PluginComposerRegistry* delegates composer registration to the Eclipse plugin mechanism. Additionally, a component file loader (*ResourceFileLoader*) that builds upon the Eclipse resources plugin — the Eclipse Platform's resource management — is implemented[2].

4.3 EMF Code And Parser Generation

The common way to generate code from a core-model is the application of a *generator model*. A generator model wraps an arbitrary core-model and needs some additional information for the code generation (e.g. a location for the generated code). For the language plugin generation, the additional information is either pre-defined in the implementation or can be derived from the language core-model. Therefore generator model usage is accomplished within the *ModelCodeGenerator*, completely invisible to the user.

EMF code generation is based on *templates*, which can be varied, making it very flexible and configurable. However, almost no modification to the provided templates is necessary. All Java class generation is left unchanged. Merely the templates for the *plugin.xml* and *MANIFEST.MF* file (which together make up the plugin's description) need slight changes to define the correct plugin dependencies and to register the *fileparser* extension to the E-CoMoGen core plugin.

The parser generation proves itself to be more tricky. The parser generator of choice is ANTLR. It has a long history and a stable implementation. It has its own metalanguage to define grammars and behaviors of scanners and parsers . The notation for the grammar rules is very similar to the EBNF notation. Is it straight forward to write an ANTLR grammar based on a language core-model. The *AntlrParserGenerator* does

[1]For EBNF the mapping is implemented by the EBNF mapping plugin as described in Section 3.1.2.

[2]In a non-Eclipse environment, E-CoMoGen extensions could be configured in a property file. An alternate composer registry could use information from the property file to retrieve composers and a component file loader based on the java.io package could be employed.

that and then feeds the grammar to the ANTLR tool, which generates Java classes out of it.

Unfortunately, ANTLR can only process LL(k) grammars and therefore not every arbitrary context free grammar. Since there are no such restrictions to language core-models (or to a corresponding EBNF grammar), the parser generation is not successful for every language core-model. Errors, resulting from improper grammars, reported by the ANTLR tool, are passed back to the user via FileProcessingExceptions. Still, it is desirable to support parser generation for more language core-models. Therefore, the ANTLR dependencies are kept loose (by using interfaces like *IParserGenerator*), making an exchange of the parser generator possible.

The process of language plugin generation based on an EBNF grammar is illustrated in Figure 4.2.

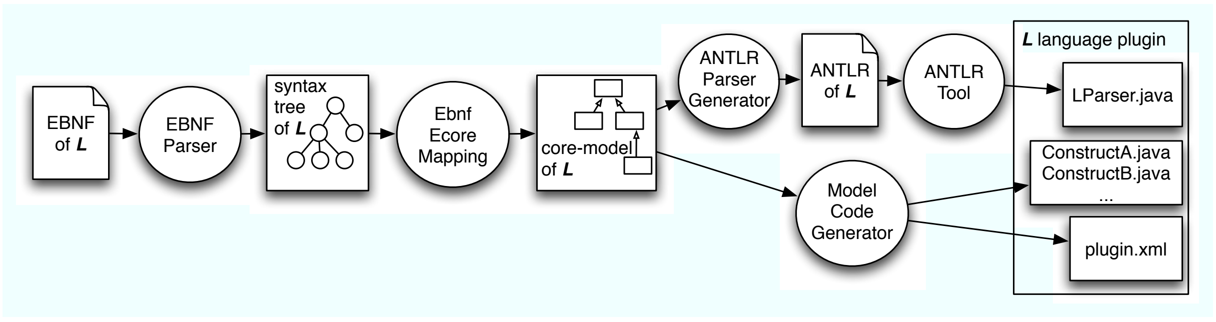


Figure 4.2: From an EBNF grammar file to a language plugin.

4.4 Hooking Into Eclipse — The E-CoMoGen IDE

While the general management of resources is provided by the Eclipse Platform, extension points allow to specify special treatments of resources. All resources exist inside the *workspace*. The workspace contains *projects* which again contain *folders* and *files*. Folders and files inside a project can be given special treatment as defined by the project's *natures*. The *ECoMoGenNature* defines one folder for composition programs, one for fragment components and one where the composed result are printed-back to. The *ResourceComponentFileLoader*, for instance, is configured with the fragment-component-folder as the location where to look for component files. The composition-program and print-back-folder are used by the *ECoMoGenBuilder*.

When a project is built[3] which has the *ECoMoGenNature* configured, the *ECoMoGen-Builder* sends an *ECoMoGenFileVisitor* to every file in the composition-program-folder. The visitor determines the language the file is written in by examining the file's extension. It then creates (by using the corresponding parser factory) and calls a parser, passes the syntax tree to the *CompositionExecutor* and prints back the result to a file in the print-back-folder. Figure 4.3 illustrates the process.

Errors which occur in form of *FileProcesingExceptions* are caught and resource-markers (a concept to add additional information to resources) containing the problem's information are attached to the erroneous input file. Resource-markers can easily be used to visualize the occurred problems in a text editor or a file hierarchy.

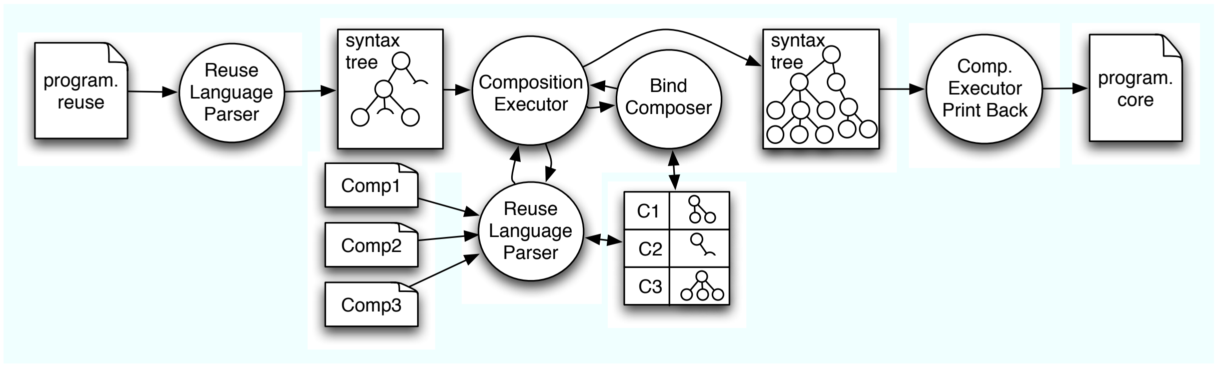


Figure 4.3: Composing a reuse language composition program and fragment components to a core language program.

The basic functionality of invasive composition is implemented and integrated into an easy-to-use IDE. However, the IDE could provide much more functionality and support for writing components and composition programs. The Eclipse Platform offers a large number of extension points which can be extended to accomplish new features with little effort. The implementation provided so far is a good starting point for further development (see Section 6.4).

The next chapter will demonstrate the tool on examples.

[3]A project can be build either automatically (when configured) or by selecting the *build* command from the Eclipse Platform's menu. Building a project means that all builders which are configured for that project are called. The JDT builder for instance compiles Java source files, located in source folders of a Java project, to class files.

5 Language Examples

To show the advantages of extending languages with invasive software composition constructs, as implemented in the E-CoMoGen tool, this chapter examines the extension of three languages. The first language — the programming language Graal introduced in Section 3.4 — will help to understand the benefits of invasive composition in general and in the domain it was initially applied — programming languages. The second chapter will deal with the XML query and transformation language Xcerpt [27] and prove that ISC can be applied to formal languages of other domains. Finally, EBNF will be considered as a core language and advantages of introducing ISC concepts to metalanguages will be demonstrated. All grammars and examples presented in this chapter work with the current E-CoMoGen implementation.

5.1 An Imperative Programming Language — Graal

The concrete syntax of the language Graal is defined by the following EBNF grammar:

```
program             =  "program", "{", declaration list, instruction, "}";


declaration list    =  { declaration };

declaration         =  type, variable, ";";

type                =  "boolean" | "integer";
instruction         =  assignment | compound | conditional | loop;
expression          =  constant | variable | binary;
variable            =  identifier;

assignment          =  variable, ":=", expression, ";";
compound            =  "begin", { instruction }, "end";
conditional         =  "if", expression, instruction, [ "else", instruction ];
loop                =  "loop", "while", expression, instruction;

constant            =  integer constant | boolean constant;
boolean constant    =  "true" | "false";
integer constant    =  ?( ('0'..'9')+ )?;

binary              =  "(", expression, operator, expression, ")";
operator            =  boolean op | relational op | arithmetic op;
boolean op          =  "&&" | "||";
relational op       =  "<" | "<=" | "==" | "!=" | ">=" | ">";
arithmetic op       =  "+" | "-" | "*" | "/";

identifier          =  ?( ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* )?;
```

5.1.1 Introducing A Component Model

The first thing we would like to add to the language is a component model — the possibility to define slots and hooks. We decide to allow slot declarations for *types*, *instructions*, *expressions* and *variables*. Hooks can be declared for *declarations* and for *instructions*. For obvious reasons, identical notations are used: For slots the scheme is `<< slot-id : slot-type >>`. For hooks it is `<+ hook-id : hook-type +>`.

```
type slot          =  "<<", (?slotid?, identifier), ":", "type",        ">>";
instruction slot   =  "<<", (?slotid?, identifier), ":", "instruction", ">>";
expression slot    =  "<<", (?slotid?, identifier), ":", "expression",  ">>";
variable slot      =  "<<", (?slotid?, identifier), ":", "variable",  ">>";

declaration hook   =  "<+", (?hookid?, identifier), ":", "declaration", "+>";
instruction hook   =  "<+", (?hookid?, identifier), ":", "instruction", "+>";
```

To integrate the new constructs in the language, some parts of the core language grammar have to be altered, in order to allow slot or hook declarations as alternatives to the corresponding concrete constructs.

```
type               =  "boolean" | "integer" | type slot;
instruction        =  assignment | compound | conditional | loop | instruction slot;
expression         =  constant | variable | binary | expression slot;
variable           =  identifier | variable slot;
```

A type, for example, can now either be *boolean*, *integer* or generic (*type slot*). A variable name can be left open by using a *variable slot*.

```
declaration list   =  { declaration | declaration hook };
```

A *declaration list* consists of an arbitrary number of *declarations*. It is now allowed to add *declaration hooks* to such a list. Note that this modification to the language is only possible because of the collection-like structure of a *declaration list*, which allows for extension. The same applies to *compounds* with respect to *instructions*.

```
compound           =  "begin", { instruction | instruction hook }, "end";
```

5.1.2 Defining A Composition Language

There are severals ways the composition language of a reuse language may look like. One important thing to consider is, how deep the constructs to execute a composition (composer calls and fragment box declarations) should integrate with the constructs used to write components. They can be kept almost independent but may as well be deeper integrated. Both approaches have their advantages. This section will introduce a composition language which constructs are more independent, while Section 5.1.4 will enhance it for deeper integration with other constructs of the reuse language.

```
composition program = box declaration list, composition list, (program bind | program);

box declaration list = {box declaration};
box declaration    =  (?boxtype?, box type), (?boxid?, identifier), ":=",
                        (?boxlocation?, path), ";";

```

```
box type           =  "program" | "declaration" | "type" | "instruction" |
                       "expression" | "variable";
path               =  ?( ('/'('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')+)+ )?;

composition list   =  { slot bind | hook extend };
slot bind          =  (?composer?, ?bind?), "bind", (?box?, identifier), ".",
                       (?slot?, identifier), "with" ,(?value?, identifier), ";";
hook extend        =  (?composer?, ?extend?), "extend", (?box?, identifier), ".",
                       (?hook?, identifier), "with" ,(?value?, identifier), ";";

program bind       =  (?composer?, ?bind?), "bind", "inline", (?value?, identifier), ";"
```

A *composition program* has two main parts: A *box declaration list*, where the fragment components are defined and a *composition list*, where composition commands are issued. `box-type box-id := box-location;` is the format of a *box declaration*, where the possible types are the ones supported by our component model and the location is a path pointing at a fragment file. A composition list consists of *slot bind* (`bind box.slot with value;`) and *hook extend* (`extend box.hook with value;`) composer calls, which can be used to compose a tree of fragment components. The root of this tree should be a program component, which has to be specified at the end of a composition program by `bind inline program-component;`.

To allow the writing of a composition program as an alternative to a normal program, the first lines of the language grammar have to be adjusted.

```
reuse program      =  program | composition program;
program            =  "program", "{", declaration list, instruction, "}";
```

5.1.3 A First Composition Example

To demonstrate the new variability of the extended Graal language, we write a program which compares three variables and neither define the values nor the type of the variables.

```
program {
   <<myType:type>> a;
   <<myType:type>> b;
   <<myType:type>> c;

   boolean result;
   <+ declarations:declaration +>

  begin
    a := <<valueA:expression>>;
    b := <<valueB:expression>>;
    c := <<valueC:expression>>;

    result := ((a == b) == c);
    <+ instructions:instruction +>
  end
}
```

To fill the slots, two sets of components are defined and two different composition programs, one using the *integer* and one using the *boolean* components, reuse the comparison program. Additionally, they extend the program to calculate the negative result.

/typeInteger.comp

```
integer
```

/expInt1.comp

```
(1 + 2)
```

/expInt2.comp

```
((10 - 9) + (1 + 1))
```

/expInt3.comp

```
3
```

/negResDecl.comp

```
boolean negResult;
```

```
program myProg := /compareThree.comp;
type theType := /typeInteger.comp;

expression exp1 := /intExp1.comp;
expression exp2 := /intExp2.comp;
expression exp3 := /intExp3.comp;

declaration negResDecl := /negResDecl.comp;
instruction negRes := /negResult.comp;

bind myProg.myType with theType;

bind myProg.valueA with exp1;
bind myProg.valueB with exp2;
bind myProg.valueC with exp3;

extend myProg.declarations with negResDecl;
extend myProg.instructions with negRes;

use myProg;
```

⇓⇓

```
program {
  integer a;
  integer b;
  integer c;

  boolean result;
  boolean negResult;

  begin
    a := (1 + 2);
    b := ((10 - 9) + (1 + 1));
    c := 3;

    result := ((a == b) == c);
    negResult := (result != true) ;
  end
}
```

/typeBoolean.comp

```
boolean
```

/expBool1.comp

```
(true && (fale || true))
```

/expBool2.comp

```
true
```

/expBool3.comp

```
((true || false) || (true && false))
```

/negResult.comp

```
negResult := (result != true) ;
```

```
program myProg := /compareThree.comp;
type theType := /typeBoolean.comp;

expression exp1 := /boolExp1.comp;
expression exp2 := /boolExp2.comp;
expression exp3 := /boolExp3.comp;

declaration negResDecl := /negResDecl.comp;
instruction negRes := /negResult.comp;

bind myProg.myType with theType;

bind myProg.valueA with exp1;
bind myProg.valueB with exp2;
bind myProg.valueC with exp3;

extend myProg.declarations with negResDecl;
extend myProg.instructions with negRes;

use myProg;
```

⇓⇓

```
program {
  boolean a;
  boolean b;
  boolean c;

  boolean result;
  boolean negResult;

  begin
    a := (true && (false || true));
    b := true;
    c := ((true||false) || (true&&false));

    result := ((a == b) == c);
    negResult := (result != true) ;
  end
}
```

5.1.4 An Enhanced Composition Language

The composition language defined so far allows the retrieval and composition of ready made components. A way to make the reuse of components more easy understandable when writing a new program, is to allow the direct binding of fragments into the program upon program writing using in-line template expansion (see Section 2.2.2).

We would like to enhance our composition language to support in-line template expansion for *instructions*, *expressions* and *variables*. Therefore, we define three new constructs (the *use* keyword indicates an inline-template expansion operation):

```
instruction bind   =  (?composer?, ?bind?), "use", (?value?, identifier), ";";
expression bind    =  (?composer?, ?bind?), "use", (?value?, identifier);
variable bind      =  (?composer?, ?bind?), "use", (?value?, identifier);
```

Again, we have to extend rules from the original grammar. A composer call for in-line template expansion may occur everywhere, where a corresponding slot is allowed to occur (because a fragment will be bound to the position of the composer call).

```
instruction = assignment|compound|conditional|loop| instruction slot | instruction bind;
expression         =  constant | variable | binary | expression slot | expression bind;
variable           =  identifier | variable slot | variable bind;
```

The Graal grammar together with the introduced alterations and the newly defined constructs constitute the Reuse-Graal language, which grammar can be found in Appendix B.

5.1.5 A Second Composition Example

The following example makes use of the enhanced composition language. A program is written, which reuses pre-define components directly.

/a.comp

```
a
```

/square.comp

```
<<result:variable>> := (<<factor:expression>> * <<factor:expression>>);
```

/b.comp

```
b
```

/cubicle.comp

```
<<result:variable>> := ((<<factor:expression>> * <<factor:expresion>>)
                                                * <<factor:expresion>>);
```

```
variable x1 := /a.comp;
variable x2 := /b.comp;

instruction calc := /square.comp;
expression exp2 := /intExp2.comp;

bind calc.factor with x2;
bind calc.result with x1;

program {

  integer use x1;
  integer use x2;

  integer c;

  begin
    use x2 := 4;

    use calc;

    c := (use x1 + use exp2);
  end
}
```

$\Rightarrow$

```
program {

  integer a;
  integer b;

  integer c;

  begin
    b := 4;
    a := (b * b);

    c := (a + ((10 - 9) + (1 + 1)));
  end
}
```

One can assign the *cubic* component to the *calc* fragment box instead of the *square* component to change the calculation.

The examples in this section showed how invasive software composition is applied in a programing language. Although, the language dealt with is not for real-life use, the concepts demonstrated in the examples can easily be ported to other languages. Generic types (Example 5.1.3), for instance, increases the reuse factor of components of any core language, which does not support it itself, remarkably. Support for in-line template expansion (Example 5.1.5) to embed small components, like algorithms, directly into code can speed up development, while adding flexibility in form of easy component exchange. The next chapter proves that this ideas can indeed be applied to a real-life language.

5.2 A Language From The Semantic Web Domain — Xcerpt

Xcerpt [27] is a XML query and transformation language, currently under development, that is used, among others, in the semantic web domain. One of the main differences with other XML query languages is that Xcerpt separates query-constructs, which defining the querying, from construct-constructs, which construct the output, in rules.

The following Xcerpt rule consists of a query part (FROM ...) that extracts information about books from a library and a construct part (CONSTRUCT ...) that assembles the results.

```
CONSTRUCT
  result {
    all result {
      var Title ,
      all var Author
    }
  }

FROM
  in {
    resource { "http://bib-lib.com" } ,
    bib {{
      book {{
        title [var Title] ,
        authors {{
          author [var Author]
        }}
      }}
    }}
  }

END
```

In the next section we will enhance the Xcerpt language with slot declarations and a small composition language to demonstrate how invasive composition could work on rules and help to make rules reusable.

5.2.1 Reuse-Xcerpt

A Reuse-Xcerpt query, which extracts information from a library, is given as a query component (*/BibQuery.comp*). It defines some open slots for variables:

```
  in {
    resource { "http://bib-lib.com"" } ,
    bib {{
      book {{
        title [
          <<title>>
        ] ,
        authors {{
          author [
            <<author>>
          ]
        }}
      }}
    }}
  }
```

To allow the binding of queries and variables, we make minimal extension to the Xcerpt language and are able to write the following composition program, which retrieves the query component, binds the open variable slots and then binds the component itself by in-line template expansion.

```
VARBOX title { /VarTitle.comp }

VARBOX author { /VarAuthor.comp }

QUERYBOX bibquery { /BibQuery.comp }

BIND bibquery.title { title }
BIND bibquery.author { author }

CONSTRUCT
  result {
    all result {
      bind var title ,
      all
        bind var author
    }
  }

FROM
   bind query bibquery

END
```

The full grammar of the extended Xcerpt can be found in Appendix C. The constructs newly introduced in the reuse language are:

```
165 var slot    = "<<", (?slotid?, identifier), ">>";
166
167 var box     = (?boxtype?, ?variable?), "VARBOX", (?boxid?, identifier),
168                     "{", (?boxlocation?, path), "}";
169 query box   = (?boxtype?, ?query?),  "QUERYBOX", (?boxid?, identifier),
170                     "{", (?boxlocation?, path), "}";
171
172 global bind = (?composer?,?bind?), "BIND", (?box?,identifier), ".", (?slot?,identifier),
173                     "{", (?value?, identifier), "}";
174 var bind    = (?composer?, ?bind?), "bind","var",   (?value?, identifier);
175 query bind  = (?composer?, ?bind?), "bind","query", (?value?, identifier);
176
177 path        =  ?( ('/'('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')+)+ )?;
```

Altered original language constructs are:

```
3 construct query rule = "CONSTRUCT", ct term, "FROM", query, "END"
4                 | "CONSTRUCT", ct term, "END"
5                 | "GOAL", goal head, "FROM", query, "END"
6                 | "GOAL", goal head, "END"
7                 | var box
8                 | query box
9                 | global bind;
```

```
29 variable = ("var", identifier) | var slot | var bind ;
```

```
99  query = real query | query bind;
100 real query =  ...
```

Although the extension of the language was kept small here, it indicates how invasive composition can be applied to a querying language. Extending the language further will not be problematic.

5.3 Towards Language Composition — EBNF

Since we are now able to extend all languages describable with EBNF, we can also extend EBNF itself. This makes it possible to structure large grammars in components and reuse parts of grammars. The constructs to declare slots and hooks used in the Reuse-Graal language for instance, are all very similar and may even be reused in other language extensions. Therefore, we will extend EBNF to Reuse-EBNF and use it to compose a modularized grammar of Reuse-Graal. EBNF is extended with a component model and a composition language, comparable to the Graal extension, by introducing new rules and altering some existing ones. The Reuse-EBNF grammar is based on the EBNF grammar used in E-CoMoGen, which can be found in Appendix A.

```
reuse ebnf       = {syntax};

syntax           = rule list | meta identifier box | rule list box | definition box |
                     bind | extend | syntax rule hook | syntax rule extend;
rule list        = syntax rule or comment, {syntax rule or comment};
syntax rule or comment  =  syntax rule | comment ;

comment            =  ?( "(*" (~( '*' | ')' | '(' ))* "*)" )?;
syntax rule        =  (meta identifier | meta identifier slot | meta identifier bind),
                       "=", definitions list, ";";
meta identifier    = ident;
ident              = ?( ('A'..'Z' | 'a'..'z')('A'..'Z' | 'a'..'z' | '0'..'9' | ' ')* )?;

definitions list  =  single definition, {("|", single definition) | syntactic primary |
                        definition hook | definition extend};

single definition =  syntactic primary, {",", syntactic primary};

syntactic primary =  optional sequence | repeated sequence | grouped sequence |
                        meta identifier | quote first | quote second | special sequence |
                        definition slot | definition bind;

optional sequence =  "[", definitions list, "]";
repeated sequence =  "{", definitions list, "}";
grouped sequence  =  "(", definitions list, ")";
special sequence  =  ?( "?" (~('?'))* "?" )?;

quote first       =  ?( '\"' (~('\"'))* '\"' )?;
quote second      =  ?( '\'' (~('\''))* '\'' )?;

syntax rule hook  =  "<+", (?hookid?, ident), "+>";
definition hook   =  "|", "<+", (?hookid?, ident), "+>";
definition slot   =  "<<", (?slotid?, ident), ">>";
meta identifier slot =  "<<", (?slotid?, ident), ">>";

rule list box     =  (?boxtype?, ?syntax?), "(","rules" ,")", (?boxid?, ident),
                        "=", (?boxlocation?, path), ";";
definition box    =  (?boxtype?, ?syntactic primary?),"(", "def",")", (?boxid?, ident),
                        "=", (?boxlocation?, path), ";";
meta identifier box = (?boxtype?, ?meta identifier?), "(", "id", ")", (?boxid?, ident),
                        "=", (?boxlocation?, path), ";";

path              =  ?( ('/'('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')+)+ )?;

definition bind       =  (?composer?, ?bind?),"->", (?value?,ident);
meta identifier bind  =  (?composer?, ?bind?),"->", (?value?,ident);
```

```
definition extend     =  "|",(?composer?, ?extend?),"->", (?value?,ident);
syntax rule extend    =  (?composer?, ?extend?),"->", (?value?,ident), ";";

bind                  =  (?composer?, ?bind?), "(","bind",")", (?box?, ident),
                          ".", (?slot?, ident), "=" ,(?value?, ident), ";";
extend                =  (?composer?, ?extend?), "(","extend",")", (?box?, ident),
                          ".", (?hook?, ident), "=" ,(?value?, ident), ";";
```

5.3.1 Reuse-Graal Composed

We prepare the Graal grammar from Section 5.1 to make it extensible by introducing hooks to some choice constructs, which allows to add more options.

```
declaration list   =  { declaration | <+declarationchoice+> };
```

```
type               =  "boolean" | "integer" | <+typechoice+>;
instruction        =  assignment | compound | conditional | loop | <+instructionchoice+>;
expression         =  constant | variable | binary | <+expressionchoice+>;
variable           =  identifier | <+variablechoice+>;
```

```
compound           =  "begin", { instruction | <+compoundchoice+> }, "end";
```

Then we define two components as reusable skeletons for the slot and hook declarations, and one which contains the composition language introduced in Section 5.1.2 and Section 5.1.4.

/slotdeclaration.ecomp

```
<<metaidentifier>>  =  "<<", (?slotid?, identifier), ":", <<type>>,">>";
```

/hookdeclaration.ecomp

```
<<metaidentifier>>  =  "<+", (?slotid?, identifier), ":", <<type>>,"+>";
```

/compositionlang.ecomp

```
composition program = box declaration list, composition list, (program bind | program);

box declaration list = {box declaration};
box declaration    =  (?boxtype?, box type), (?boxid?, identifier), ":=",
                       (?boxlocation?, path), ";";

box type           =  "program" | "declaration" | "type" | "instruction" |
                       "expression" | "variable";
path               =  ?( ('/'('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')+)+ )?;

composition list   =  { slot bind | hook extend };
slot bind          =  (?composer?, ?bind?), "bind", (?box?, identifier), ".",
                       (?slot?, identifier), "with" ,(?value?, identifier), ";";
hook extend        =  (?composer?, ?extend?), "extend", (?box?, identifier), ".",
                       (?hook?, identifier), "with" ,(?value?, identifier), ";";

program bind       =  (?composer?, ?bind?), "use", (?value?, identifier), ";";
instruction bind   =  (?composer?, ?bind?), "use", (?value?, identifier), ";";
expression bind    =  (?composer?, ?bind?), "use", (?value?, identifier);
variable bind      =  (?composer?, ?bind?), "use", (?value?, identifier);
```

We can now assemble the Reuse-Graal grammar by writing a composition program that extends the original Graal grammar and merges it with the composition language grammar component. The contents of the additional small components, used in the composition program, correspond to the components' names.

```
reuse program     =  program | composition program;

(rules) graalcore      = /graal.ebnf;

(id) type slot         = /typeslot.ecomp;
(id) instruction slot = /instructionslot.ecomp;
(id) expression slot  = /expressionslot.ecomp;
(id) variable slot    = /variableslot.ecomp;
(id) declaration hook = /declarationhook.ecomp;
(id) instruction hook = /instructionhook.ecomp;

(extend) graalcore.typechoice        = type slot;
(extend) graalcore.instructionchoice = instruction slot;
(extend) graalcore.expressionchoice  = expression slot;
(extend) graalcore.variablechoice    = variable slot;
(extend) graalcore.declarationchoice = declaration hook;
(extend) graalcore.compoundchoice    = instruction hook;
-> graalcore;


(rules) slot1                = /slotdeclaration.ecomp;
(def)   type                 = /type.ecomp;
(bind)  slot1.type           = type;
(bind)  slot1.metaidentifier = type slot;
-> slot1;

(rules) slot2                = /slotdeclaration.ecomp;
(def)   instruction          = /instruction.ecomp;
(bind)  slot2.type           = instruction;
(bind)  slot2.metaidentifier = instruction slot;
-> slot2;

(rules) slot3                = /slotdeclaration.ecomp;
(def)   expression           = /expression.ecomp;
(bind)  slot3.type           = expression;
(bind)  slot3.metaidentifier = expression slot;
-> slot3;

(rules) slot4                = /slotdeclaration.ecomp;
(def)   variable             = /variable.ecomp;
(bind)  slot4.type           = variable;
(bind)  slot4.metaidentifier = variable slot;
-> slot4;

(rules) hook1                = /hookdeclaration.ecomp;
(def)   declaration          = /declaration.ecomp;
(bind)  hook1.type           = declaration;
(bind)  hook1.metaidentifier = declaration hook;
-> hook1;

(rules) hook2                = /hookdeclaration.ecomp;
(bind)  hook2.type           = instruction;
(bind)  hook2.metaidentifier = instruction hook;
-> hook2;

(rules) compoisitionlang     = /compositionlang.ecomp;
-> compoisitionlang;
```

6 Outlook

Many ideas for the modeling of languages originate from the ones presented in [6]. However, some language modeling approaches and their possible consequences for this work have not been taken into account. This chapter will take a short look at the relations of these approaches to this work and point out where future work could be directed to integrate them.

It was suggested earlier, to extend the implementation of E-CoMoGen with additional features. Some proposals about how this extensions might look like and how they can be supported by the Eclipse Platform are presented in Section 6.4.

6.1 Semantics Of Languages

In [6] static semantics of languages are described by introducing new binary relations between constructs. A similar approach could be used in language core-models: Additional references between construct-classes could be introduced to denote semantic relationships (e.g. every variable use has a reference to a variable declaration where the variable names are equal). After parsing a program to a syntax tree, this references can be filled by rewriting the tree to a graph. For this, one could apply a Java based rule engine (e.g. Jess [28]) and use EMF's validation framework to check if a program is semantically correct.

Even if invasive software composition works without knowing about the specifics of the language's semantics, the benefits of such support should be discussed. Describing all the semantics of a core language is a tedious task and much more complex than describing its syntax. One could avoid this by using existing tools, like interpreters and compilers, to find syntactic errors in core language programs. The additional semantics in reuse languages can be (and are already) checked during composition. These semantics, which are based on the ISC concepts, are either generic for all reuse languages, or can be checked in the implementation of a composer operation. An example of such a semantic constraint is that certain slots have to be bound during a composition (E-CoMoGen checks this kind of semantics and reports an error if such a constraint is violated).

A more interesting issue for further research is the semantics of fragment components. A fragment component on its own is may not be semantically correct with respect to the core language. However, a composition of fragment components results in a program of the core language, which has to conform to the semantics of the core language. If it does not, it is sometimes not clear if the mistake was made during composition, already during component definition or if it is the result of syntax issues[1].

If semantics, of which kind whatsoever, should be defined they have to be described somehow. It should be possible to extend EBNF further to integrate descriptions of language semantics directly into EBNF grammars. These kind of grammars are likely to get complex. Invasive composition of language grammars (Example 5.3) might help here to structure grammars into components. Developing a graphical editor which allows conformable definition of complex language core-models might also be another option. A mapping from EBNF would then become unnecessary and the strong similarities between language-core models and the EBNF language can be reconsidered.

6.2 Automatic Component Model Generation

Universal genericity and *universal extensibility* were introduced and defined in Section 2.2.1 as concepts, which, when present in a language, make the language well suited for component definition. ISC can easily be used to simulate these two concepts by *automatic component model generation* [6, 29]. The principle is, to derive slot and hook declaration constructs automatically for all existing constructs in a core language.

The following algorithm can be applied to a language core-model to add a corresponding slot declaration construct for every construct in the language and simulate *universal genericity*:

- For each language construct-class named <*X*> (which is not a sub-construct)

1. create a construct-class <*X*>_*Slot*.
2. make <*X*>_*Slot* a subclass of *LCSlotDeclaration*.
3. create a construct-class <*X*>_*SlotChoice*
4. make <*X*>_*SlotChoice* a subclass of *LCChoice*.
5. make <*X*> and <*X*>_*Slot* subclasses of <*X*>_*SlotChoice*.
6. replace all references to <*X*> by references to <*X*>_*SlotChoice*.

To simulate *universal extensibility*, hook declaration constructs have to be added to a language everywhere, where collection-like constructs (which correspond to EBNF's repeated sequences) occur. An algorithm of the following kind is sufficient:

[1]A problem which occurs in invasive source code composition is naming. If, for instance, each of two components declare and use a local variable with the same name, the compiler for the core language will assume that both components refer to the same variable after the components are invasively composed, which is not the desired result. In such a case, one variable should be renamed automatically by the composition tool.

- For each reference in a construct-class (which subclasses *LCAggregat*) of multiplicity *any* to a construct-class *<X>*

1. create a construct-class *<X>_Hook*, if it does not exit already.
2. make *<X>_Hook* a subclass of *LCHookDeclaration*.
3. create a construct-class *<X>_HookChoice*, if it does not exist already.
4. make *<X>_HookChoice* a subclass of *LCChoice*.
5. make *<X>* and *<X>_Hook* subclasses of *<X>_HookChoice*.
6. replace the references to *<X>* by a references to *<X>_HookChoice*.

These algorithms are easy to implement and were prototypical tested with E-CoMoGen on the Graal language. However, the concrete syntax of slot and hook constructs needs to be defined somehow. It can not be easily generated automatically — at least not without language dependent configuration of the component model generator. Additionally, the extended language still needs some composition language to make use of the component model, which has to be defined manually. A tool suite for component model generation which is configurable and also supports the introduction of composition language constructs to a reuse language is thinkable.

Another way to accomplish a supported component model extension of a language could be language grammar composition as indicated in Example 5.3.

6.3 The Relationship Between Reuse And Core Language

A reuse languages handled in E-CoMoGen is always a superset of the corresponding core language (see Figure 3.7). This is one of the fundamental ideas of the composition by tree rewriting. If the core language was not a subset of the reuse language, rewriting a reuse language tree could never result in a tree containing core language constructs not belonging to the reuse language.

It might be desirable to have a *multi-stage programming* [30] emphasized approach of multi-stage invasive composition, where the composition of reuse language components results in components of another reuse language and so forth until components of one reuse language are eventually composed to core language programs. That allows the definition of components of different granularity. The first reuse language could be an *Architecture Description Language*, while the last may allow fine grained adjustments to small components. In such a multi-stage composition, the principle of letting every reuse language employed be a superset of the next reuse language employed is hard to maintain.

6.4 Further Development Of E-CoMoGen

The current implementation of E-CoMoGen is working fine on executing the composition, but could use enhancement on the user interface. Some ideas which came up during the development, but are not yet implemented, are sketched in this section.

- **Projects and folders**: A *project*, configured with the E-CoMoGen *nature*, is a container for fragments and composition programs. As described in Section 4.4, folders for different purposes exist in such a project. It would be desirable to have a *wizard* to create an E-CoMoGen project (like it exists for Java projects) and an appropriate *property page* to define folders for the different purposes.
- **Text editor**: A text editor with *syntax highlighting* is a must-have for an IDE. Syntax highlighting rules can be derived from a language core model. A configuration, allowing the user to define colors and text appearances, should be provided. An *outline view*, which is a structured graphical representation of source code, could also be provided for composition programs and fragments.
- **Error feedback**: Composition errors are already marked when using the Eclipse standard text editor. However, this process can be improved. Especially errors occurring inside a fragment which is loaded into a fragment box are hard to visualize. Some kind of *composition preview* could be implemented to give a view on the partial, but erroneous, composition result.
- **Semantic check**: As indicated before, semantic errors of core-language programs can be checked by existing tools (Section 6.1). How exactly such tools may be integrated in E-CoMoGen should be explored. Defining reuse languages based on Java and integrating E-CoMogen with the Eclipse JDT may be an interesting case-study for that purpose.
- **Language enhancement tool suite**: Tools to support the definition of reuse languages can also improve the usability of E-CoMoGen. It was already mentioned that *automatic component model generation* could be integrated by providing such tools (Section 6.2).

Ideas for other useful features for E-CoMoGen may surface by looking at other Eclipse IDEs or by examining the offered extension points of various plugins which belong to the Eclipse Platform.

A question that has to be answered when designing a new feature is, whether the feature can be implemented to work for every (generated) language plugin or if additional code generation should be provided. The question might be, for example, if there should be a text editor for E-CoMoGen in general or one generated for every reuse language.

7 Conclusion

Relationships between different *systems* were discussed and illustrated (mostly using megamodels) throughout this thesis. To draw a first conclusion and to give a final overview about how the different systems relate to each other, a megamodel is presented in Figure 7.1, which concentrates on all important systems introduced and their relations. It unites the megamodels presented in earlier chapters.

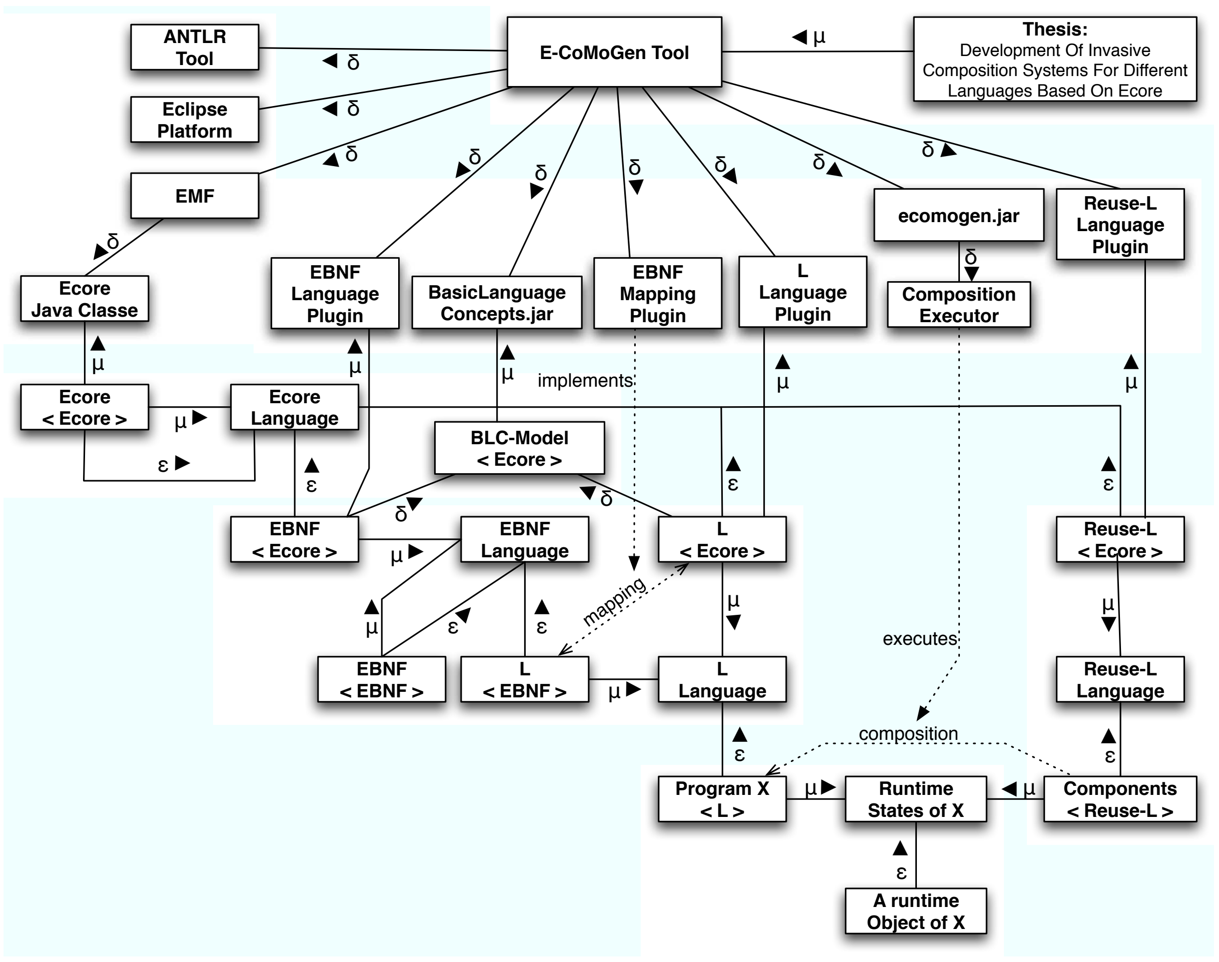


Figure 7.1: The megamodel of all systems this thesis is about. The language L can be substituted by any core language, while Reuse-L can be substituted by any reuse language which is an extension of L.

This thesis showed how ideas about language modeling have been successfully employed in the E-CoMoGen tool. Languages can be modeled in MOF or Ecore consistently by introducing a common upper layer to all language models. Ideas from two-layer language modeling and concepts from EBNF were successfully reused.

Invasive software composition constructs were introduced into several existing languages and can now be introduced into others as well using E-CoMoGen. The implementation of a composition tool which can work on any formal language underscores the universality of the ISC technique. It allows ISC (and the tool implementing it) to exist outside the scope of a concrete language, which is a great advantage especially in new domains, like the semantic web, where languages are under development and in continuous change, and where it is unforeseeable which languages will make it to wide-spread applications.

For implementation oriented modeling, the Eclipse Modeling Framework and its metamodel Ecore proved to be the right choice. The code generated based on models is well structured and easy and flexible to use in applications, without requiring complicated adjustments or configurations to the code generation facilities.

The Eclipse Platform keeps its promises and offers an ideal foundation for an integrated development environment for composition programs and fragment components, as well as for the writing of reuse language grammars to integrate support for new languages in form of language plugin generation. The tool is kept extensible using the Eclipse Platform's plugin mechanism making extensions (e.g. in form of new composers) easy. However, it was possible to keep the tool's core Eclipse independent, allowing it to be used easily in other environments (e.g. as a command-line tool for scripting).

The E-CoMoGen tool is a successful result of this work. Development of the tool should be continued to make its use more user friendly. A tool, which people without deeper knowledge about language modeling can use to introduce invasive composition concepts to any language they like, should be a desired goal. This can be efficiently accomplished by extending existing features of the Eclipse Platform.

Concerning language modeling, future work can benefit from the principle of two-layered language models, but should reconsider the structure of the upper layer. It should investigate deeper into the modeling of language semantics with Ecore and the distinction between concrete and abstract syntax.

A EBNF Grammar Of EBNF

```
syntax              =  { syntax rule or comment };

syntax rule or comment  =  syntax rule | comment;
comment             =  ?( "(*" (~( '*' | ')' | '(' ))* "*)" )?;
syntax rule         =  meta identifier, "=", definitions list, ";";

meta identifier     =  ?( ('A'..'Z' | 'a'..'z')('A'..'Z' | 'a'..'z' | '0'..'9' | ' ')* )?;

definitions list    =  single definition, {"|", single definition};

single definition   =  syntactic primary, {",", syntactic primary};

syntactic primary   =  optional sequence | repeated sequence | grouped sequence |
                       meta identifier | quote first | quote second | special sequence;

optional sequence   =  "[", definitions list, "]";
repeated sequence   =  "{", definitions list, "}";
grouped sequence    =  "(", definitions list, ")";
special sequence    =  ?( "?" (~('?'))* "?" )?;

quote first         =  ?( '\"' (~('\"'))* '\"' )?;
quote second        =  ?( '\'' (~('\''))* '\'' )?;
```

B EBNF Grammar Of Reuse-Graal

```
reuse program       =  program | composition program;
program             =  "program", "{", declaration list, instruction, "}";

declaration list    =  { declaration | declaration hook };

declaration         =  type, variable, ";";

type                =  "boolean" | "integer" | type slot;
instruction = assignment|compound|conditional|loop| instruction slot | instruction bind;
expression          =  constant | variable | binary | expression slot | expression bind;
variable            =  identifier | variable slot | variable bind;

assignment          =  variable, ":=", expression, ";";
compound            =  "begin", { instruction | instruction hook }, "end";
conditional         =  "if", expression, instruction, [ "else", instruction ];
loop                =  "loop", "while", expression, instruction;

constant            =  integer constant | boolean constant;
boolean constant    =  "true" | "false";
integer constant    =  ?( (’0’..’9’)+ )?;

binary              =  "(", expression, operator, expression, ")";
operator            =  boolean op | relational op | arithmetic op;
boolean op          =  "&&" | "||";
relational op       =  "<" | "<=" | "==" | "!=" | ">=" | ">";
arithmetic op       =  "+" | "-" | "*" | "/";

identifier          =  ?( (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’_’|’0’..’9’)* )?;

type slot           =  "<<", (?slotid?, identifier), ":", "type",        ">>";
instruction slot    =  "<<", (?slotid?, identifier), ":", "instruction", ">>";
expression slot     =  "<<", (?slotid?, identifier), ":", "expression",  ">>";
variable slot       =  "<<", (?slotid?, identifier), ":", "variable",  ">>";

declaration hook    =  "<+", (?hookid?, identifier), ":", "declaration", "+>";
instruction hook    =  "<+", (?hookid?, identifier), ":", "instruction", "+>";

composition program = box declaration list, composition list, (program bind | program);

box declaration list = {box declaration};
box declaration     =  (?boxtype?, box type), (?boxid?, identifier), ":=",
                        (?boxlocation?, path), ";";
box type            =  "program" | "declaration" | "type" | "instruction" |
                        "expression" | "variable";
path                =  ?( (’/’(’a’..’z’|’A’..’Z’|’_’|’.’|’0’..’9’)+)+ )?;
composition list    =  { slot bind | hook extend };
slot bind           =  (?composer?, ?bind?), "bind", (?box?, identifier), ".",
                        (?slot?, identifier), "with" ,(?value?, identifier), ";";
hook extend         =  (?composer?, ?extend?), "extend", (?box?, identifier), ".",
                        (?hook?, identifier), "with" ,(?value?, identifier), ";";
program bind        =  (?composer?, ?bind?), "use", (?value?, identifier), ";";
instruction bind    =  (?composer?, ?bind?), "use", (?value?, identifier), ";";
expression bind     =  (?composer?, ?bind?), "use", (?value?, identifier);
variable bind       =  (?composer?, ?bind?), "use", (?value?, identifier);
```

C EBNF Grammar Of Reuse-Xcerpt

The concrete syntax of the slightly extended Xcerpt (Reuse-Xcerpt) is derived from the grammar included in [31].

```
program = construct query rule, { construct query rule } ;

construct query rule = "CONSTRUCT", ct term, "FROM", query, "END"
                | "CONSTRUCT", ct term, "END"
                | "GOAL", goal head, "FROM", query, "END"
                | "GOAL", goal head, "END"
                | var box
                | query box
                | global bind;

goal head = ct term
                | "out", "{", out resource, ",", ct term, "}"
                | "out", "[", out resource, ",", ct term, "]" ;

number terminal = ?( (’0’..’9’)+ )?;
identifier  = ?( (’A’..’Z’ | ’a’..’z’) (’A’..’Z’ | ’a’..’z’ | ’0’..’9’ | ’_’ | ’-’)* )?;
string quote = ?( ’\"’ (~(’\"’))* ’\"’ )? ;
comp        = ?( "<" | ">" | "<=" | ">=" | "!=" | "=")? ;

float number = number terminal,[".", number terminal ] ;

out resource = "resource", "{", resource, "}"
                | "resource", "[", resource, "]" ;
in resource  = out resource
                | ("or" | "and"), "{", out resource, {",", out resource}, "}"
                | ("or" | "and"), "[", out resource, {",", out resource}, "]" ;
resource     = [ ( string quote | variable ), "," ], string quote ;

variable     = ("var", identifier) | var slot | var bind ;

label list   = identifier, {",", identifier} ;

ct term      = identifier, [ordered ct list | unordered ct list]
                | identifier, "@", ct term
                | "optional", ct term, ["with", "default", ct term]
                | "(", ct term, ")" ;

ct subterm   = identifier, [ordered ct list | unordered ct list]
                | string quote
                | variable
                | "^", identifier
                | identifier, "@", ct subterm
                | "optional", ct subterm, [ "with", "default", ct subterm ]
                | "(", ct subterm, ")" ;

ct subterm coll = "all", ( ct subterm
                | "[", ct subterm list, "]"
                | "{", ct subterm list, "}" ),
                [ ordering | grouping ]
                | "some", amount, ( ct subterm
                | "[", ct subterm list, "]"
```

```
               | "{", ct subterm list, "}" ),
               [ ordering | grouping ]
               | "(", ct subterm coll, ")" ;

ordering     = "order", "by", [identifier], "[", label list, "]" ,
               ["ascending" | "descending"] ;

grouping     = "group", "by", "[", label list, "]" ;

amount       = number terminal
               | "-", number terminal
               | number terminal, "-", number terminal ;

ordered ct list = "[",  [ct subterm list], "]" ;
unordered ct list = "{", [ ct subterm list], "}" ;
ct subterm list = [ct attributes], ( ct subterm | ct subterm coll ),
               { ",", ( ct subterm | ct subterm coll ) } ;

ct attributes = "attributes", "{", ct attribute, { ",", ct attribute }, "}"
               | "(", ct attributes, ")" ;
ct attribute = identifier, ( "[", (string quote | variable), "]"
               | "{", (string quote | variable), "}" )
               | variable
               | "^", identifier
               | identifier, "@", ct attribute
               | "all", ( ct attribute | "[", ct attribute,
                     { ",", ct attribute }, "]"), [ ordering | grouping ]
               | "some", amount, (ct attribute | "[", ct attribute,
                     { ",", ct attribute }, "]"), [ ordering | grouping ]
               | "(", ct attribute, ")" ;

function     = identifier;

fun param    = ct subterm
               | ct subterm coll
               | ct attributes ;

condition    = variable, ( identifier | comp ), cond param
               | cond param, ( identifier | comp ), variable
               | ( identifier | comp ), "(", variable, ",", cond param, ")"
               | ( identifier | comp ), "(", cond param, ",", variable, ")"
               | ( "and" | "or" | "not" ), "[", condition, { ",", condition }, "]"
               | ( "and" | "or" | "not" ), "{", condition, { ",", condition }, "}";

cond param   = qr subterm
               | function ;

query        = real query | query bind;
real query   = ( qr term
               | ( "and" | "or" ), "{", query, { ",", query }, "}"
               | ( "and" | "or" ), "[", query, { ",", query }, "]"
               | "not", query
               | "in", "{", in resource, ",", query, "}"
               | "in", "[", in resource, ",", query, "]"
               | "Fail"
               | "(", query, ")"
           ),  ["where", "{", condition, "}"] ;

qr term      =  identifier, [ ordered qr list | unordered qr list]
               | variable, ["->", qr term]
               | identifier, "@", qr term
               | "optional", qr term
               | "desc", qr subterm
```

```
                | "(", qr term, ")" ;

qr subterm    = identifier, [ordered qr list | unordered qr list]
                | number terminal
                | float number
                | variable, ["->", qr subterm]
                | "^", identifier
                | identifier, "@", qr subterm
                | ( "desc" | "optional" | "without" | "position",
                             ( number terminal | variable ) ), qr subterm
                | "(", qr subterm, ")" ;

ordered qr list = "[", [  qr subterm list ], "]"
                | "[[", [ qr subterm list ], "]]" ;
unordered qr list = "{", [  qr subterm list], "}"
                | "{{", [ qr subterm list ], "}}" ;

qr subterm list = [qr attributes],  qr subterm, { ",", qr subterm } ;

qr attributes = "attributes", "{", qr attribute, { ",", qr attribute }, "}"
                | "attributes", "{","{", qr attribute, { ",", qr attribute }, "}","}"
                | ( "desc" | "optional" | "without" ), qr attributes
                | "(", qr attributes, ")" ;
qr attribute = identifier, ( "[", ( identifier | variable ), "]" |
                             "{", ( identifier | variable ), "}" )
                | identifier, "{{", "}}"
                | variable, ["->", qr subterm]
                | "^", identifier
                | identifier, "@", qr subterm
                | ( "desc" | "without" | "optional" ), qr attribute
                | "(", qr attribute, ")" ;

dt label      = identifier;

dt term       = [ identifier, "@" ], dt label,
                (ordered dt list | unordered dt list);

dt subterm    = dt term
                | number terminal
                | float number
                | "^", identifier
                | identifier, "@", dt subterm ;

ordered dt list = "[", [dt subterm list], "]" ;
unordered dt list = "{", [dt subterm list], "}" ;
dt subterm list = [dt attributes], dt subterm, { ",", dt subterm } ;
dt attributes = "attributes", "{", dt attribute, { ",", dt attribute }, "}" ;
dt attribute = dt label, ( "[", string quote, "]" | "{", string quote, "}" );


var slot     = "<<", (?slotid?, identifier), ">>";

var box      = (?boxtype?, ?variable?), "VARBOX", (?boxid?, identifier),
                    "{", (?boxlocation?, path), "}";
query box    = (?boxtype?, ?query?),  "QUERYBOX", (?boxid?, identifier),
                    "{", (?boxlocation?, path), "}";

global bind = (?composer?,?bind?), "BIND", (?box?,identifier), ".", (?slot?,identifier),
                    "{", (?value?, identifier), "}";
var bind     = (?composer?, ?bind?), "bind","var",   (?value?, identifier);
query bind   = (?composer?, ?bind?), "bind","query", (?value?, identifier);

path         =  ?( ('/'('a'..'z'|'A'..'Z'|'_'|'.'|'0'..'9')+)+ )?;
```

Bibliography

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

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

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

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

[5] Object Management Group. Unified modeling language — UML. `http://www.uml.org/`, May 2006.

[6] Torsten Bürger. Contributions to language composition using standard semantic web techniques. Master's thesis, Dresden University of Technology, October 2005.

[7] The Eclipse Foundation. Eclipse platform technical overview, April 2006.

[8] The Eclipse Foundation. Eclipse project. `http://www.eclipse.org/`, May 2006.

[9] The Eclipse Foundation. Eclipse modeling framework — EMF. `http://www.eclipse.org/emf/`, May 2006.

[10] Object Management Group. Meta object facility — MOF. `http://www.omg.org/mof/`, May 2006.

[11] TU Dresden Software Engineering Group. E-CoMogen webpage. `http://web.inf.tu-dresden.de/~jh30/work/rewerse/comogen`, May 2006.

[12] Jean-Marie Favre. Foundations of model (driven) (reverse) engineering : Models – episode i: Stories of the fidus papyrus and of the solarus. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/13`.

[13] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/21`.

[14] Jean-Marie Favre. Megamodelling and etymology. In Jean Bezivin and Reiko Heckel, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2006. `http://drops.dagstuhl.de/opus/volltexte/2005/427`.

[15] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, 2003.

[16] Jean Bézivin, Vladan Devedzic, Dragan Djuric, Jean-Marie Favreau, Dragan Gasevic, and Frédéric Jouault. An m3-neutral infrastructure for bridging model engineering and ontology engineering. In *Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'2005)*, pages 159–171. Springer-Verlag, 2005.

[17] The REWERSE project. *Report on the design of component model and composition technology for the Datalog and Prolog variants of the REWERSE languages*, August 2004.

[18] The REWERSE project. *Composition of Rule Sets and Ontologies*, May 2006.

[19] O. Lehrmann Madsen, B. Möller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, Reading, MA, 1993.

[20] The Eclipse Foundation. Eclipse java development tools — JDT. `http://www.eclipse.org/jdt`, May 2006.

[21] The Eclipse Foundation. Eclipse plugin development environment — PDE. `http://www.eclipse.org/pde`, May 2006.

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

[23] Anna Gerber and Kerry Raymond. Mof to emf: there and back again. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 60–64, New York, NY, USA, 2003. ACM Press.

[24] Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland, Mar 2004.

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

[26] Terence Parr. ANTLR — another tool for language recognition — parser generator. `http://www.antlr.org/`, May 2006.

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

[28] Sandia National Laboratories. Jess, the rule engine for the java platform. `http://herzberg.ca.sandia.gov/jess`, May 2006.

[29] The REWERSE project. *Prototype component models and composition technology toolset for integration of logic-programming-like REWERSE languages*, September 2005.

[30] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[31] Clemens Ley. A robust parser for the web query language xcerpt, July 2004. project work.

Confirmation

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

Dresden, June 8, 2006