Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2010-12

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Declarative Rule-Based Composition, Validation, and Improvement of EBNF Grammars for the Management of Language Extensions

Svetoslav Mihaylov

am Institut für

Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

7. September 2010

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 7. September 2010

Bachelor's Thesis

Declarative Rule-Based Composition, Validation, and Improvement of EBNF Grammars for the Management of Language Extensions

Svetoslav Mihaylov

7. September 2010

Supervised by Prof. Dr. phil.-nat. Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science
Georg-August-Universität Göttingen

Abstract

Nowadays there often exists the need to adapt a language to address the needs of a specific area of application. Language extensions provide a way to add this functionality conditionally; while preserving the core of the language intact. In this thesis we present a methodology that supports the work with language syntax extensions in the form of EBNF grammars. We introduce two new grammar types that can make the work with extension grammars automated and yet customizable. We also explore several methods that ease the maintenance of EBNF grammars. Finally, we disuss algorithms that support the proposed methodology and present them in the context of a prototype implementation for the Eclipse Platform.

Acknowledgements

First, I would like to thank Prof. Dr. Jens Grabowski for the supervision of my thesis, as well as for offering me this opportunity in the first place. I also want to thank Dr. Benjamin Zeiss for his continuous helping and advising me. I would also like to thank Vladislav Todorov and Philipp Kiefer, who helped me with the proofreading. Finally, I want to thank my family for their endless support and encouregement throughout my studies.

Thank you all!

Contents

1 Introduction **7**
1.1 Related Work 9
1.2 Contributions 11
1.3 Thesis Structure 11

2 Foundations **12**
2.1 (E)BNF Grammars and (E)BNF Extensions 12
2.2 Platform and Plug-ins Used 14
2.2.1 The Eclipse Platform 14
2.2.2 Xtext and the Eclipse Modelling Framework (EMF) 16
2.2.3 BNFTools 16
2.2.4 ANTLR 17
2.2.5 Xpand 17

3 Extensible EBNF Grammars **19**
3.1 Composition of Grammars 19
3.1.1 Delta Grammars 21
3.1.2 Merge Grammars 22
3.1.3 Composition Operators 24
3.2 Validating and Transforming Extensible EBNF Grammars 26
3.2.1 Validation after a Core Grammar Update 26
3.2.2 Rule Inlining 27
3.2.3 EBNF to ANTLR Conversion 28

4 Implementation **29**
4.1 Using BNFTools 29
4.2 Conversion to Delta Grammars 31
4.3 Creation of Merge Grammars 32
4.4 Generation of a Composite EBNF Grammar 33
4.5 Other Features 35
4.5.1 Validation after a Core Grammar Update 36
4.5.2 Rule Inlining 36
4.5.3 EBNF to ANTLR Conversion 37

5 Summary and Outlook **38**
5.1 Summary 38
5.2 Outlook 39

Acronyms

ABNF Augmented Backus Naur Form

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

BNF Backus Naur Form

DSL Domain-Specific Language

EBNF Extended Backus Naur Form

ECL Extensible Computer Language

EMF Eclipse Modelling Framework

ETSI European Telecommunications Standards Institute

ETSIBNF European Telecommunications Standards Institute's BNF

IDE Integrated Development Environment

IETF Internet Engineering Task Force

ISO International Organization for Standardization

JDT Java Development Tools

JSE Java Syntactic Extender

MPS Meta Programming System

MWE Modeling Workflow Engine

PDE Plug-in Development Environment

SDF Syntax Definition Formalism

SQL Structured Query Language

TTCN-3 Testing and Test Control Notation Version 3

UML Unified Modelling Language

URI Unique Resource Identifier

XML eXtensible Markup Language

1 Introduction

Formal languages and their descriptions are the foundation of computer programming languages. Their efficient specification and transformation into machine code has been a popular research area in computer science for more than 50 years. Nowadays, with the advent of domain-specific languages, the interest in intuitive and error-free language syntax specification has been renewed. One of the most popular specification methods in this area are Backus Naur Form (BNF) grammars — a concept introduced in 1959 by John Backus and later refined by Peter Naur [3].

BNF grammars play a vital part in the language creation process. They provide an intuitive way of describing the context-free grammars. Furthermore, the creation of a parser based on a BNF grammar is usually automated and thus they play an integral role in compiler creation. Practically all computer languages are defined with BNF grammars or variants (such as Extended Backus Naur Form (EBNF)) and often these grammars are publicly available. An example is the Testing and Test Control Notation Version 3 (TTCN-3) [9], where the EBNF is a part of the standard.

Currently one of the most popular test specification languages, TTCN-3 is a worldwide standard applicable to all kinds of computer-related testing. However, the necessity sometimes exists to adapt the language specification so that it better fulfills the needs of a particular software engineering problem — for example performance or real-time testing [9]. This is done via small grammars, called extensions or packages, which complement the core TTCN-3 grammar by providing new rules as well as extended versions of the current rules. Figure 1.1 illustrates this idea.

The process of extending a BNF grammar is the cause of the main problem we discuss in this thesis: the so-called composition of grammars, that is, the creation of new combined BNF grammars from a core grammar and one or more of its extensions. Manually creating composite grammars for large language specifications (such as TTCN-3) can be very hard and error-prone, but so far there exist no practical and convenient solutions (see Section 1.1). Furthermore, it is unclear what happens when several extensions are to be merged together, especially if both redefine the same rules. The three grammars below illustrate these issues.

The core grammar defines a simple class definition statement that is then extended with an optional **interface** part and a required **domain** part. Both extensions redefine the same rule (**start**) from the core grammar and add elements to it at the same place (**modify_table** and **select**). The way in which the two extensions are to be merged is not clear; there

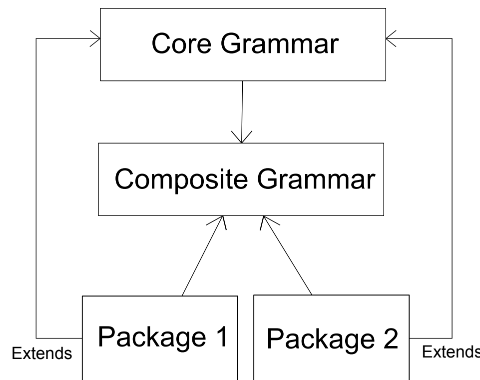

Figure 1.1: The Composition of Grammars

exist several possbilities, such as concatenating the added elements or inserting them as alternatives. We address this ambiguity of the composition process by discussing several composition operators to specify how two (or more) such extensions are to be handled.

```
1 grammar SQL_Definition;
2
3 start ::= create_table
4 create_table ::= ("CREATE") ID...
```

Listing 1.1: Core Grammar Example

```
1 grammar SQL_Extension;
2
3 import "SQL_Definition.bnf"
4
5 start ::= create_table | modify_table
6 modify_table ::= insert | update | delete
7 ...
```

Listing 1.2: Extension Grammar 1 Example

```
1 grammar SQL_Access;
2
3 import "SQL_Definition.bnf"
4
5 start ::= create_table | select
6 select ::= ("SELECT") ("*" | ID) ("FROM") ...
```

Listing 1.3: Extension Grammar 2 Example

In this thesis we are going to present a methodology that makes the composition process automated, as well as tooling that supports it. As a basis for the implementation we used an Eclipse plug-in called BNFTools, which has been developed at the University of Göttingen. It provides tools for the general work with and improvement of EBNF grammars

and is implemented for the Eclipse Platform with the help of Xtext, the Eclipse Modelling Framework (EMF), and Xpand, which are also used for the implementation in this thesis. For our solutions, we added several new features to this plug-in, while enhancing some of the old ones. We also extended the grammar syntax so that it can support two newly introduced grammar types: the *delta* grammars and the *merge* grammars.

Delta grammars offer a different notation for extension grammars. The rules in these delta grammars can be separated in two kinds: regular rules, that is the rules not found in the original grammar and extension rules that only consist of the changes compared to the original rule, as well as their exact positions. Delta grammars provide an overview of the changes made by the package and can be likened to the concept of patches in Unix, that is, updates that only contain the changes related to the original file. Delta grammars can be automatically created from an extension grammar. They also serve as an intermediate step to the second newly introduced language: the merge grammar.

A merge grammar combines all the information from one or more extensions that is needed for their merging with the core grammar. The main purpose of a merge grammar is to concretely specify the behavior of a grammar composition. This is needed to solve ambiguities that naturally arise when merging a core grammar with several of its extensions. To that extent we introduce four so-called *composition operators* that specify how the extensions interact with each other. These operators are created by default in the merge grammars and the user can modify them globally or locally in order to change the specification of the composition.

With the help of these two grammars, we are able to make the composition of grammars a fully automated four-stage process. We also discuss several miscellaneous topics that relate to the improvement and management of EBNF grammars, including rule inlining and EBNF to ANTLR conversion.

1.1 Related Work

The main problem addressed in this thesis is the lack of tools to support the composition of grammars. In this section we are going to present several projects also related to language syntax extensions and discuss their similarities and differences compared to our thesis. There have been two different approaches to the topic: one that focuses on specific extensible languages and one that supports their creation in general.

The first ideas about programming languages that allow users to extend their syntax come from the sixties and seventies, for example the Extensible Computer Language (ECL) [35] (features extensible data types) and ALGOL with run-time extensions [6] (uses EBNF to create new data structures and constructs). Recently the topic was revived with the occurrence of several new extensible languages; a good overview is at [37]. Some of these languages approach syntax extensions in the same way as we do in this thesis: for exam-

ple, IMP [5; 14] is a language whose syntax can be extended by creating new grammar rules, loosely similar to the EBNF rules. In contrast to our approach that is suitable for extending any language defined in an EBNF grammar, each of these languages defines its own constructs and ways to be extended, leading to a need for specific tooling for each.

More important to the thesis, however, is the second methodology, which is concerned with the creation and support of extensible languages in general. We are going to present four related projects: Silver [34], the Syntax Definition Formalism (SDF) [33], xPico [11], and Meta Programming System (MPS) [16]. Silver is an example of a platform that uses attribute grammars [21] as a means to specify languages and their extensions. Syntax extension in Silver is done with a so-called collection operator, which adds new attribute definitions to an existing attribute. This platform has already been used for specifying and extending several languages (for example Java in [36]), but it makes the work with simple EBNF grammars unnecessarily complicated.

SDF [33] is another language specification framework whose main purpose is to support modular context-free grammars; that is grammars that are specified independently, but can extend each other and be used together. The creation of modular grammars is supported by the SDF syntax, which, while similar to EBNF, provides a number of additional features such as disambiguation constructs. SDF allows the concatenation of several grammar modules and combines mutual rules as alternatives. The disambiguation constructs, for example priorities and restrictions, can be used to quickly select from or discard these alternatives. Overall, SDF facilitates can be used for the work with language extensions, but the composition options are limited and it requires manual grammar conversion when used for existing languages.

Another similar project is [11]. It presents a tool for the management of DSLs in the form of small core languages that can be easily extended. A framework called xPico is introduced that supports the process of creating and extending language defined in EBNF-like grammars. These grammars offer several new constructs, among which configurable rules. Configurable rules are specified in the core grammar and provide the extension points of the language. An extension grammar is then a combination of reconfigured core rules and new EBNF rules, very similar to delta grammars in this thesis. In xPico, however, the focus lies on the core grammars, which have to be created with the purpose of being extensible. Furthermore, as in SDF, manual conversion to the xPico syntax is needed, which can be troublesome due to the need to specify the extension points of the language.

Finally, the project MPS [17] from Jetbrains is a language-oriented software development environment that supports the creation and management of language extensions. In it, languages are defined and stored as abstract syntax trees. Languages can extend other arbitrary languages, which allows them to use or override nodes from the original language. Overall, the framework is flexible and the concepts are similar to the ideas in this thesis; the main difference is that instead of EBNF, MPS utilizes abstract syntax trees.

These were four projects that were very close to the topic of this thesis. Each of them

provides a different possibility for language syntax extensions and while each of them has their strengths, none are convenient for working with simple EBNF grammars and grammar extensions.

1.2 Contributions

The main focus of this thesis is on the presentation of a methodology for the automated composition of grammars and their extensions. We present the theory behind our approach, as well as some algorithms to support the concrete implementation. We also introduce two new Domain-Specific Language (DSL)s, the delta and merge grammars, that can also be used outside the scope of the thesis. Besides the composition of grammars, we also discuss several further topics: finding inconsistencies in extensions when the core grammar is updated, converting a grammar from the EBNF syntax to the ANTLR notation, and validation and quickfix support for inlining EBNF rules. Our results in each topic were implemented as a plug-in for the Eclipse Platform.

1.3 Thesis Structure

The rest of the thesis is structured as follows: in Chapter 2 we present the foundations of the thesis: BNF grammars, the Eclipse Platform, and the plugins used for the implementation (including BNFTools). Afterwards, Chapter 3 focuses on the theoretical basis, with a detailed description of the problems and their solutions. It presents the composition of a language and its extensions as a four part process and includes the introduction of delta and merge grammars. Following that, Chapter 4 describes the algorithms behind the implementation as a part of BNFTools, as well as a brief guide to working with the plug-in. Afterwards, Chapter 5 presents replacement rules as a further topic, and Chapter 6 concludes the thesis.

2 Foundations

This chapters deals with the background needed to understand the thesis. We start with a description of BNF grammars and their extensions — their area of application, the reason people use them and the way they relate to our thesis. Afterwards, we describe the basis of our implementation — the Eclipse Platform and the plug-ins and tools that we used: Xtext, BNFTools, ANother Tool for Language Recognition (ANTLR), EMF, and Xpand.

2.1 (E)BNF Grammars and (E)BNF Extensions

BNF grammars are a half-a-century old, but still a widely used computer science concept. A BNF grammar is used to specify the syntax of a language and is thus the first step in the creation of a language. Afterwards, the derivation of a lexer and a parser from the grammar is needed, but this can often be done automatically with tools such as ANTLR [2]. There are plenty of examples of freely available BNF grammars for popular computer languages: the Structured Query Language (SQL) [18], Java [12], or C [30] to name a few.

Before we present the BNF form used in this thesis, we provide the formal definition of context-free grammars. A grammar can be viewed as a quadruple, consisting of a set of terminals T, a set of nonterminals N, a starting symbol S from N, and a set of production rules P. The terminals are the alphabet symbols and they only appear on the right side of rules. The nonterminals are grammar symbols that can appear on either side of the rules. Both sets have to be disjoint. The production rules are rules of the form: $A \rightarrow B$, where A is a single nonterminal, and B is a set of terminals and/or nonterminals. [1], p.42.

The notation that BNF grammars introduce is more convenient and consists merely of a set of rules, one of which is designated as the start rule. It introduces one special symbol to denote alternatives inside a rule: "'|"'. There exist many variations and extensions of BNF grammars and thus there is no single unified notation for them. Two prominent versions are Augmented Backus Naur Form (ABNF) from the Internet Engineering Task Force (IETF) [13] and EBNF from the International Organization for Standardization (ISO) [15]. The first is a standard used in Internet specifications, while the second is more universally popular and is thus the notation we use in the thesis. Both forms define similar constructs; the differences lie mainly in the notations. Below we only present the EBNF special symbols and their meaning:

- Strings, that is, character sequences in quotation marks, denote terminals

- Round parentheses denote a grouping; that is, the following two are equivalent:

```
1 RuleA ::= RuleB ( RuleC_1 RuleC_2) RuleD
2
3 RuleA ::= RuleB RuleC RuleD
4 RuleC ::= RuleC_1 RuleC_2
```

Listing 2.1: Expression EBNF

- Square brackets denote an optional element or a group of optional elements
- Curly brackets denote a repeatable group of elements, that is a multiplcity of 0..n. If there is a plus sign after then second bracket, the elements need to occur at least once (multiplicity of 1..n)
- The exclamation mark is the negation symbol: all elements except the one before the sign are accepted

In the following, we present an example of an EBNF grammar with the above syntax. *Expression* is the start rule whose right side consists of three nonterminals. An *Integer* is a digit other than zero, followed by any number of digits; an *Operator* is a plus sign. The digits and the operator symbols are the terminals.

```
1 Expression ::= Integer Operator Integer
2 Integer ::= "1".."9" {"0".."9"}
3 Operator ::= "+"
```

Listing 2.2: Expression EBNF

The main focus of this thesis, however, lies not on language specification but on language extensions in the form of EBNF grammars that extend other EBNF grammars. We will refer to the language extensions as **grammar extensions** or **packages** and to the original grammars as **core grammars**. Extending languages is a concept that can be compared to that of DSLs [20]: both are most often meant to be used in a specific field, but instead of creating new languages, grammar extensions enhance existing languages with additional functionality.

Packages are EBNF grammars that import and redefine rules from another EBNF grammar, while also introducing new ones. We will mainly be concerned with extension rules — that is, redefining rules that only add elements to the original rule without deleting anything from it. We will refer to the other kind of redefining rules as replacement rules and will discuss them at the end of the thesis, in Chapter 6.

Listing 2.2 presents an example of an extension grammar to the core grammar in Listing 2.1. Here, the extension grammar provides both some refinement (an expression can contain any number of terms and their respective operators), as well as additional functionality (the minus sign can also be chosen as an operator). While the composition of

these grammars is trivial (replace the original rules with the redefined ones), this is rarely the case for real grammars.

```
1 import "Expression.bnf"
2
3 Expression ::= Integer {Operator Integer}
4 Operator ::= "+" | "−"
```

Listing 2.3: Expression Extension.bnf

An example of a language whose syntax is defined and extended with EBNF grammars is TTCN-3 from ETSI: the main syntax is defined in its own grammar at [9] and there exist several official extensions to it [7; 8]. Other languages that have been extended in the same manner are SQL [19] and C [31]. These examples serve to illustrate that grammar extensions are frequently employed for languages that have a wide spectrum of application. Extending the core syntax of such languages would introduce unnecessary features that only certain users require. But, with the help of grammar extensions, they can still be efficiently utilized everywhere they are needed, thus eliminating the need for several different languages that possibly do similar things.

As shown in Section 1.1, there is a lack of tools that support the work with language syntax extensions in EBNF notation. The main problem is that before the packages can be used in practice, they need to be attached to the core grammar, a process that we call the *composition* (or *merging*) of grammars.

2.2 Platform and Plug-ins Used

The implementation of our prototype tool was done exclusively for the Eclipse Platform by extending the plug-in BNFTools. In this section, we are thus going to provide some background information needed to understand our implementation. We start by describing the Eclipse Platform [28] — a popular software development environment with support for several programming languages, most notably Java. Following that, there are four subsections on the most important plug-ins for Eclipse that we used: the first focuses on Xtext, a tool for the creation and management of DSLs and EMF, a framework for the work with structured models. The second subsection describes BNFTools — the plug-in we extended for the implementation, while the third is about ANTLR — an LL(*) parser generator. The last subsection presents Xpand — a language for transforming models to text.

2.2.1 The Eclipse Platform

The grammar composition tool built for this thesis is based on the Eclipse Platform. This platform is a part of Eclipse: "an open source community whose projects are focused on building an extensible development platform, runtimes and application frameworks

for building, deploying and managing software across the entire software lifecycle" [27]. Nowadays, there exist numerous projects that are part of Eclipse and we use several of them for our tooling.

The Eclipse Platform is the foundation for the Eclipse Project and represents a framework for basic software development with limited features but a lot of support for extensions, which have led to its popularity. There exist a large number of plug-ins, or bundles, that supply the platform with various additional functionality. Such plug-ins are developed and published both by Eclipse itself and by third parties. Our prototype implementation is focused on extending the BNFTools plug-in, described in Subsection 2.2.3.

The architecture of the Eclipse Platform reflects the importance that extensibility plays in it. We show an overview of the platform in Figure 2.1. The Eclipse Platform is in the middle and encompasses several different components — Platform Runtime, Workspace, Workbench, Help, and Team and Debug. These components are the core of Eclipse and everything else builds on them. The shapes outside the actual platform are its extensions, with the most important ones being the Java Development Tools (JDT) and the Plug-in Development Environment (PDE). In the following, we give a brief explanation of the parts most relevant for our tool.

- **Platform Runtime**. The Eclipse Platform Runtime was built with the idea of plug-in support. The most noteworthy feature is that plug-ins are loaded on-demand, which allows the user to install any number of them without decreasing the overall performance of the platform (lazy loading).

- **Workbench**. The Workbench provides an environment for the workspace resources, for example files. Perhaps the most important concept in it are the perspectives. A perspective provides a specific layout of the workbench, often suited for the work in a particular area of software development. For example, two of the default perspectives are the Java Perspective and the Team Synchronizer Perspective.

- **Plug-in Development Environment (PDE)**. This is the environment that provides support for the development of plug-ins for the Eclipse Platform.

- **Plug-ins**. As shown in the figure, plug-ins can exist independently, or rely on others to work.

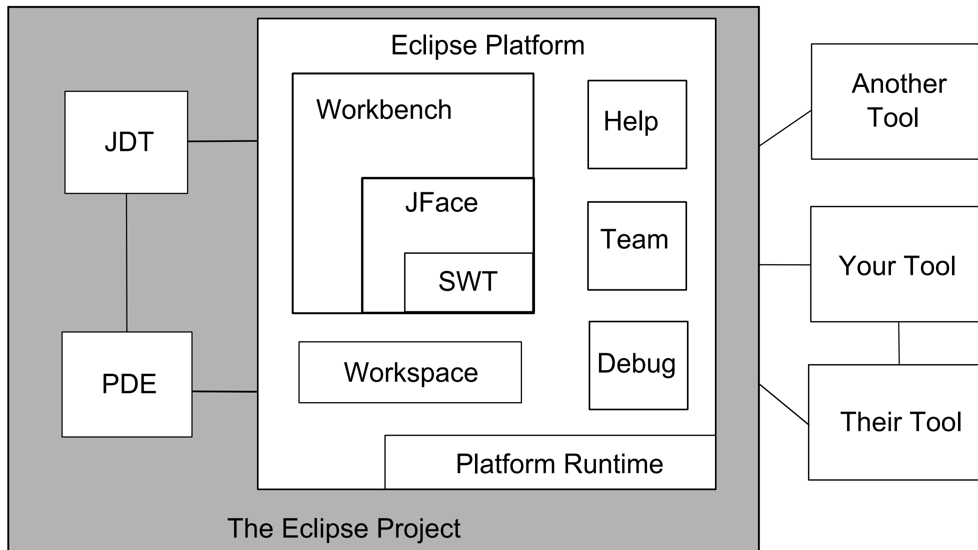

Figure 2.1: Eclipse Architecture Overview

2.2.2 Xtext and the Eclipse Modelling Framework (EMF)

Xtext [29] is an Eclipse project that was critical to our tool implementation. Xtext is a framework that specifically targets the creation of DSLs.In Xtext, a DSL is defined in an EBNF grammar, from which Xtext can derive a variety of constructs: a lexer and a parser (with the help of ANTLR), an Ecore model, a linker, a complete syntax-driven Integrated Development Environment (IDE) integrated into Eclipse, and others. The generated IDE itself can include a variety of features such as code completion and syntax coloring. Because Xtext is only important as a technical background to the implementation, we are not going to present it further. The reader may find more details in the official documentation [4].

Eclipse Modelling Framework (EMF) is an Eclipse project that was used in direct conjunction with Xtext. It provides a framework for the work with and code generation from structured models and possesses its own metamodel, called Ecore. Ecore greatly aided us in the implementation by providing an easy method for traversing the parse tree of EBNF grammars. More information can be found in [23] and [32].

2.2.3 BNFTools

BNFTools is an Eclipse plug-in that provides an IDE for the work with EBNF grammars. It is developed at the University of Göttingen and thus we were able to extend it for the implementation of our thesis. The original plug-in is based on Xtext by specifying the EBNF

syntax in an Xtext grammar, which Xtext used for the creation of the IDE. The functionality of BNFTools encompasses creating, validating, and refactoring EBNF grammars, and it provides several tools for that, such as syntax coloring, railroad diagrams, quick fixes, name refactorings, and so on. To implement the solutions we present in the thesis, we introduce a number of additional features to the plug-in.

2.2.4 ANTLR

ANTLR [2] is a tool for the generation of lexers and parsers from EBNF-like grammars and Java-like constructs for semantic actions. While not part of the Eclipse Project, ANTLR has been fully integrated into the Eclipse Platform by means of a third-party plug-in. The syntax of an ANTLR grammar (one that uses only the basic features of the language) is very similar to an EBNF syntax, so here we are only going to outline the differences. The functionality of ANTLR, however, extends far beyond its EBNF constructs. For example the user may insert Java code inside the grammar, or they may specify the construction of a syntax tree. Details can be found in [10] or [22].

The differences between the ANTLR and the EBNF notation presented before are several. Optional and repeated sequences are represented via the cardinality operators **?** and *. Terminal rules have to start with a capital letter, while nonterminal rules have to start with a small letter. Strings are specified with single apostrophes, instead of doubles. The start of the right side of a rule is denoted by a colon and its end by a semicolon.

2.2.5 Xpand

Xpand is a language that supports model-to-text transformations [25; 27]. Its main construct is the DEFINE-block that represents a single template for one class of the input model. In every such template, there can be various other statements, including ones calling other templates and ones that print output into a file. We are now going to present the main constructs of the language with the help of the example in Listing 2.3. There are two syntactical peculiarities of Xpand: statements of the language have to be written inside the so called guillemets; keywords are written in capital letters.

```
1  «IMPORT ebnf»
2
3  «DEFINE Main FOR EtsiBnf»
4      «EXPAND Main FOREACH Rule»
5  «ENDDEFINE»
6
7  «DEFINE Main FOR Rule»
8      «FILE name.rule»
9  Class «FILE name» {
10     //TODO: Autogenerated Class
11 }
12     «ENDFILE»
13 «ENDDEFINE»
```

Listing 2.4: Xpand Example Grammar

IMPORT Provides a way to refer to a given model without always having to specify the full classpath. In the example above, it has a similar meaning to "import ebnf.*" in Java.

DEFINE This is a template, which consists of a name, possible parameters, and the metamodel class for the template. Afterwards, there can be several statements until it is closed with an *ENDDEFINE* statement. Templates are invoked by their name and support polymorphism in the model class they are defined for: in the above example there are two *DEFINE* blocks with the same name that differ in their respective classes.

EXPAND This statement expands a *DEFINE* block for the given element(s), inserts its input on that spot, if any, and then proceeds. There are two possible operators: *FOR* for single elements and *FOREACH* for lists. It is also possible to specify aliases for the elements (keyword *AS*) or use iterators (keyword *ITERATOR*). In the example, the template for an EBNF grammar expands the template for each of the grammar's rules.

FILE This statement is used for creating and writing into files. By default a new file is created each time *FILE* is called, possibly erasing old versions. The statement allows any expression to be used as a name, with "/" used to specify subdirectories. The file mode is closed via *ENDFILE*. Everything except Xpand statements is printed exactly as written, including whitespaces and newlines.

The configuration for model generations or model transformations is specified in the Modeling Workflow Engine (MWE) syntax. MWE itself is meant to support the execution of Eclipse components as a so-called workflow [24]. A workflow is generated automatically for every new Xpand project and can be adapted for the specific needs of a project.

3 Extensible EBNF Grammars

This chapter presents our approach to handling language extensions. We start off with a detailed description of the main problem and our solution. We then present two new grammar types and explore the customization options when composing grammars.

3.1 Composition of Grammars

The main problem we address is the lack of methodology and tools that support the composition of EBNF grammars. We present an approach that allows automation, while also presenting several means to exactly specify the composition to avoid ambiguities. We divide the whole process in four distinct steps. Each of these stages presents the core grammar and its packages in a different way and together they give the user a better overview and environment for the management of language extensions. The following diagram presents the four stages of the composition process:

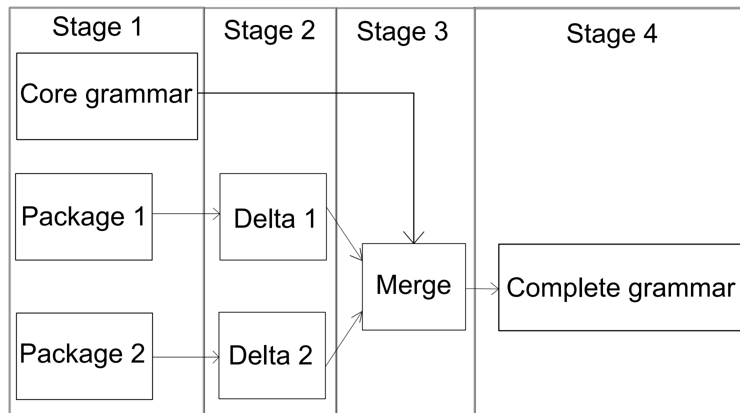

Figure 3.1: The Composition of Grammars

The first stage is trivial. It consists of just a core grammar and one or more of its extensions. All the grammars should be in EBNF form. There is one construct added to the EBNF syntax that provides the ability to import other EBNF grammars. An imported grammar can be concretely specified as either a core grammar, a package, or an updated core gram-

mar. Grammar extensions import the core grammar and define or overwrite EBNF rules from the core grammar.

The second stage deals with the conversion of the packages into delta grammars. This grammar type, detailed in Subsection 3.1.1, has been created with the goal of having a clear notation for language extensions. Thus, a delta grammar allows the user to instantly see the changes the package makes to the core grammar, that is what rules were added, which ones were extended, and what the extensions are. Writing extensions in this form, however, (or their manual conversion to it) would be tedious because of the need to specify the exact extension positions. Hence, an important part of the implementation dealt with the automatic conversion of a package to a delta grammar.

The third step involves the creation of a merge grammar from a core grammar and its extensions (in delta form). The merge grammar presents the extensions to be merged in an easy to read file. We use these grammars as a way to deal with the problem that the composition of several packages and a single core grammar is often ambiguous. For that, we present several composition operators that allow users to concretely define the behavior of the composition.

The last stage consists of the single EBNF grammar that contains both the core grammar rules (extended or not) and the package rules.

The remainder of this Section deals with the detailed concepts that we introduced for this composition process: delta grammars, merge grammars, and the composition operators. In the following there are three small EBNF grammars that we are going to use as examples throughout the chapter. The core grammar defines a simple class definition statement that is then extended with an optional *interface* part and a required *visibility* part.

```
1  grammar ClassDefinition;
2
3  Programm ::= Header "{" {Statement}+ "}"
4  Header ::= "class"
5  Statement ::= ...
```

Listing 3.1: Core Grammar Example

```
1  grammar Interface;
2
3  import "ClassDefinition.bnf"
4  Header ::= ["interface"] "class"
```

Listing 3.2: Package 1 Example

```
1  grammar Visibility;
2
3  import "ClassDefinition.bnf"
4  Header ::= Visibility "class"
5  Visibility ::= "public" | "private" | "protected"
```

Listing 3.3: Package 2 Example

3.1.1 Delta Grammars

A delta grammar is an EBNF-like grammar that specifies a language extension. The idea is to introduce a grammar type that is only an intermediary state as the usual EBNF syntax does not provide a precise way to specify changes to a grammar. For example, it does not differentiate between core rules, new rules, or changed rules. It also does not show where exactly a rule has changed. A delta grammar is precise in these aspects.

The overall syntax of delta grammars is very similar to that of EBNF grammars and is in fact an extension of the regular EBNF syntax. In the following, we present two examples of delta grammars, converted from the EBNF grammars in Listings 3.2 and 3.3 above.

```
1 /delta;
2
3 import "ClassDefinition.bnf" /core
4
5 Header(0) <− Visibility
6 Visibility ::= "public" | "private" | "protected"
```

Listing 3.4: Delta Example Grammars

```
1 /delta;
2
3 import "ClassDefinition.bnf" /core
4 Header(0) <− ["interface"]
```

Listing 3.5: Delta Example Grammars

In order to specify delta grammars as simply as possible, we add two new elements to the BNF syntax. The first is the **grammar type** that can be set at the start of a grammar as either "/delta" or "/bnf". If none is used, "/bnf" is assumed and thus denoting a regular EBNF notation. The second new element are the extension rules that can be used as an alternative to the regular EBNF rules.

The "/delta" keyword extension rules are allowed in the grammar. Regular EBNF rules remain a valid part of a delta grammar, however they are only used for package rules that do not extend a core rule, that is, rules that are completely defined in the package. Extension rules are, as the name suggests, rules that extend rules from the core grammar. The extension rules are the core of a delta grammar as they compactly present the position at which the original rule is meant to be extended as well as the complete extension. The structure of an extension rule is as follows:

```
1 ExtensionRule ::= ID "(" INT ")" "<−" {Atom}+ ["?"]
```

Listing 3.6: Extension Rule Syntax

ID is the name of the rule (referring to a rule name in the core grammar) and has the same syntax as in EBNF, but one big semantical difference: the names of extension rules

need no longer be unique. This is a consequence of the fact that a package rule can extend the original rule in several positions.

INT is an integer that gives the exact index position of the extension in the original rule. The counter excludes the left side of the rules. It starts at zero and each element counts as one — literals, rule references, special symbols.

"<-" is the literal we used for separating the left from the right side in an extension rule. It also provides a distinct way of differentiating extension rules from EBNF rules ("::=").

(Atom)* are the tokens and special symbols that form the extension itself. Unlike EBNF rules, an extension rule puts no constraints on the syntax of its right side. Tokens and special symbols can occur in any order. For example, "')"' is a valid right side of an extension rule.

Delta grammars present a more intuitive environment for the management of language extensions, however, they are not suitable for the actual creation of the package. On the one hand, computing extension indices against the core grammar is a cumbersome task and on the other hand, the same rule indices can become invalid if the core grammar changes.

The most important purpose of delta grammars is the part they play in the creation of a composite grammar. It is possible to automatically convert packages to delta form and then to proceed with the creation of a merge grammar from the core and delta grammars. Delta grammars can also be used for maintaining language extensions. We provide one such example in Section 3.2.

3.1.2 Merge Grammars

Merge grammars are the second new grammar type that we introduce in our thesis. These grammars are used as a precise description of the composition of a core grammar and its packages. They present all the information about the composition in one place and allow the user to understand how the different extension grammars interact with each other. They allow the specification of the composition process via the so called *composition operators* (used to avoid ambiguities and explained in the next subsection). The following listing presents a merge grammar created from the core and delta grammars above:

```
1  /merge;
2
3  import "ClassDefinition.bnf" /core label: Core
4  import "Visibility_delta.bnf" /package label: Package1
5  import "Interface_delta.bnf" /package label: Package2
6
7  global combinator: /and
8
9
10 // rule combinator: Header /and
```

```
11 // Header ::= "class"
12
13 hook combinator: Header (0) /and
14   ( "Package1" ) // Header(0) <− Visibility
15   ( "Package2" ) // Header(0) <− ["interface"]
```

Listing 3.7: Merge Example Grammar

Merge grammars specify a grammar type at the beginning (as do delta grammars) with the keyword "/merge". We also introduce a new optional component to the import clauses, called **label**. We use this label as a concise way to refer to packages inside the merge grammar. The most important new element is a new type of rules, called merge rules. Those are the only kind of rules allowed in a merge grammar. The following listing presents the syntax of merge rules and their three subtypes in EBNF notation:

```
1 MergeRule ::= GlobalCombinator | RuleCombinator | HookCombinator
2
3 GlobalCombinator ::= "global" "combinator:" LOGIC [";"]
4
5 RuleCombinator ::= "rule" "combinator:" ID LOGIC {"(" STRING ")"} [";"]
6
7 HookCombinator ::= "hook" "combinator:" ID "(" INT ")" LOGIC {"(" STRING ")"}+ [";"]
```

Listing 3.8: Merge Rule Syntax

The syntax of a merge rule itself is simply a subdivision into the three kinds of merge rules: global, rule, and hook combinators. All merge rules share two elements: **logic**, which is used to store the composition operator of the rule (see the following Subsection) and the literal **combinator**. Because of the differences between the three types of merge rules, we will present and explain them separately, starting with the most simple one.

A **GlobalCombinator** is a merge rule that can be used only once in a grammar. It is defined by the literal **global** and its purpose is to define a single composition operator for the whole grammar. This is the weakest kind of combinator and it is overwritten by more concrete operators. It is useful for specifying a uniform composition operator that can then be overwritten at specific positions.

The second type of merge rules are the **RuleCombinators**. They are defined by the literal **rule** and are very similar to global merge rules in that their purpose is to specify a composition operator. As the name suggests, a **RuleCombinator** only works for the corresponding rule, given by the **ID** element, and it overrides the global combinator. Rule combinators present a way to precisely tune the composition of the grammars at the rule level, i.e., it allows users to have different operators for the composition without having to take care of every extension.

The third merge rule kind is the **HookCombinator**. A hook combinator exists for every rule extension in the given delta grammars. These rules provide not only a way for the user to very precisely set the composition operators, but they also play a vital role in the actual composition by combining the information given by the delta grammars in a suitable way.

This is done as follows: a single hook operator exists for every rule extension. The first part of the syntax (after the defining literal **hook**) is similar to the syntax of the extension rules in delta grammars - the **ID** is the rule name and the **INT** is the integer denoting the extension position. What follows is a set of strings that represent grammar labels, with each label corresponding to an imported delta grammar that has an extension to the current rule at the current position. The **logic** part of a hook combinator is optional and if used, it overrides previous operators.

Merge grammars are thus a succession of combinators. There exists a single global combinator, followed by a rule combinator for every redefined rule from the core grammar. Every rule combinator is itself followed by hook combinators for each of the extension positions of the rule. Like that, a merge grammar is both hard to read and possesses a large amount of information. Therefore, the global and rule combinators are always optional. We also require from the implementation that a merge grammar is commented in the following way: every **RuleCombinator** is followed by the corresponding rule from the core grammar and every **HookCombinator** is accompanied by all the extension rules that it represents. In the latter, for the user's convenience, the commented extension rule can be inserted right after the label of the package it comes from (that is, those comments are inside the merge rule itself).

A merge grammar created in this way eases the understanding of the composition that is going to take place. Most notably, it becomes very easy to notice where two or more extension grammars interact with each other. If there are such cases, merge grammars can then be used to specify the way this interaction takes place by editing the composition operators in the merge rules. Once this has been done, the creation of a complete composite grammar can easily be done automatically, as shown in the following chapter. In the following, we show an example grammar created from the merge grammar above:

```
1
2  grammar Class_definition;
3
4  Programm ::= Header "{" { Statement } + "}"
5  Header ::= Visibility [ "interface" ] "class"
6  Statement ::= ...
7
8  Visibility ::= "public" | "private" | "protected"
```

Listing 3.9: Composite Example Grammar

3.1.3 Composition Operators

The composition operators provide the user with an opportunity to precisely define the composition process by specifying the way how multiple extensions at the same rule index are inserted in the rule. Different users may have different expectations or aims when combining several packages and the composition operators provide a way to express them.

We implemented four different composition operators: the **and**, **reverse and**, **or**, and **interleave** operators. We present each of the four operators by defining its keyword and its function, and by showing how the operator works in an example. In the example, shown in Listing 3.10, *ClassDefinition* is the core grammar and *Visibility* and *Interface* are two extensions to it that both redefine the rule *Header*:

```
1 ClassDefinition:  Header ::= "class"
2 Visibility:      Header ::= Visibility "class"
3 Interface:       Header ::= ["interface"] "class"
```

Listing 3.10: A Core Rule and Its Two Extensions

- The default operator is the **AND** operator (keyword: "/and"). Intuitively, it means that if two rules extend a rule at the same position, those extensions have to be used together in a concatenated form. That is, the **and** operator simply inserts the extensions into the rule at the proper place. The order in which the extensions are inserted is taken from the import statements at the start of the merge grammar: first come extensions from the first imported package, then from the second, and so on. Here is the full rule from the example above when the **and** operator is used:

```
1   Header ::= Visibility [ "interface" ] "class"
```

Listing 3.11: The "And" Operator

- The second operator is **reverse and** (keyword: "/andr"). It works exactly like the **and** operator but reverses the order in which the extensions are inserted. It provides the user with a quick way to change the composition order without having to rearrange the import clauses manually. It is also helpful if a different order is needed for different parts of the grammar. Here is the same example as above but with **reverse and**:

```
1   Header ::= [ "interface" ] Visibility "class"
```

Listing 3.12: The "And Reverse" Operator

- The **or** operator (keyword: "/or") inserts the "|" sign between two (or more) rule extensions with the same index. In the following we present an example of the same rule as above but with the **or** operator used. A **reverse or** operator is not implemented, as it is semantically equivalent to **or**:

```
1   Header ::= (Visibility | [ "interface" ]) "class"
2   (Header ::= ([ "interface" ] | Visibility) "class")
```

Listing 3.13: The "Or" Operator

- The fourth and final operator is the **interleave** operator (keyword: "/interleave"). It combines the possibilities of all the other operators into one and inserts them as alternatives into the rule. It allows the appearance of the variable rule parts in any order and possible subsets. Following is the same example for **interleave**:

```
1  Header ::= ( Visibility | [ "interface" ] | ( Visibility [ "interface" ] ) | ( [ "interface" ] Visibility )) "class"
```

Listing 3.14: The "Interleave" Operator

The first of the two operators (**and** and **reverse and**) always work as intended, but the second two introduce some peculiarities we had to take care of and as a result, the output when using them may differ from what the user intuitively expects. The reason for this is that the usage of the operators, in the way we just explained it, does not always produce syntactically correct EBNF rules. In the following we present an example of two rules that, while similar to the ones above, produces erroneous syntax when used with the **or** operator:

```
1  ClassDefinition:  Header ::= "class"
2  Visibility:       Header ::= Visibility "class"
3  Interface:        Header ::= "interface" | "class"
4
5  //Result after using the "OR" operator:
6     Header ::= (Visibility | "interface" | ) "class"
```

Listing 3.15: A Problem when Using the "Or" Operator

This presents one of the cases where an **or** or an **interleave** operator does not produce correct output. The problems consists of there being a special symbol at the end of one of the extensions: either a "|", as in the example, or a bracket. In order to alleviate that we ignore the two operators at any extension index that ends or begins with such a special symbol and use the **and** operator instead.

3.2 Validating and Transforming Extensible EBNF Grammars

In the following we are going to discuss three topics that, while still related to EBNF grammars, focus on their improvement in general. In *Validation after a Core Grammar Update* we are going to present a method that discovers inconsistencies caused by a core grammar update and afterwards we will discuss a method for grammar refactoring called *Rule Inlining*. Finally, we will present how to convert EBNF grammars to the ANTLR notation.

3.2.1 Validation after a Core Grammar Update

New versions of a core grammar may cause various inconsistencies in its extensions. For example, core rules may have been changed, renamed, or deleted. As a result, the package

grammar may have to be changed as well. Thus, it is useful to algorithmically detect these possible inconsistencies and present the results to the language engineer. We make no attempt to automatically fix the package. While in some cases this may be possible, often the solutions are nontrivial and ambiguous.

We recognize two types of inconsistencies, defined by the type of core grammar change that caused them: change in an extended rule or a deletion of an extended rule. Below we present an example for each:

- **Change in an extended rule.** A rule extended in the package has been changed in the new core grammar.

```
1 import "ClassDefinition1.bnf" /core   //core grammar
2 import "ClassDefinition1_1.bnf" /update   //updated core grammar
3
4 Inconsistency:
5 Delta: Header(1) <− ID   //index may point to a different position
```

Listing 3.16: Example of an Inconsistency because of a Change in a Rule

- **Missing rule.** A core rule that has been referenced or extended in the package is missing from the updated grammar, either because of a deletion or renaming. If the rule with the name "Header" is redefined in a package, the meaning is changed, that is, it becomes a (possibly) undefined new rule.

```
1 "ClassDefinition.bnf": Header ::= "class"
2 "ClassDefinition1_2.bnf": HeaderA ::= "class"
```

Listing 3.17: Inconsistency after Renaming a Rule

3.2.2 Rule Inlining

As the number of rules in a grammar grows larger, its maintenance becomes increasingly harder. Often there are rules that are referenced only once, which presents the possibility for their inlining. Inlining a rule consists of taking the rule's right side and inserting it in the place of its only reference. Afterwards, the rule may be deleted.

Reducing the number of rules, however, is a two-sided coin. It makes a grammar more compact, reduces the depth of the created parse trees , and in some cases may make a grammar easier to read and maintain. Sometimes, however, the effect may be just the opposite: several huge rules are created that are very hard to depict and understand due to the large amount of tokens and alternatives they possess. For that reason, rule inlining should only be utilized with care and at the right time, for example, right before the grammar is finalized and a lexer is constructed.

3.2.3 EBNF to ANTLR Conversion

An EBNF is just a specification and does not allow language recognition by itself. A lexer and a parser are needed that recognize the language described by the grammar if a computer should detect language membership of a sentence. These tasks are often easier done by tools instead of manual implementations and for that reason we also provide a model to text conversion of EBNF grammars to the notation of one such tool: ANTLR. ANTLR provides an environment that supports all EBNF constructs, among others, and thus this conversion is lossless and purely syntactical. The conversion is thus very straightforward and will be described in detail in the following chapter.

4 Implementation

In this chapter we describe how users can interact with the plug-in and present the algorithms behind the implementation of the features introduced to BNFTools. We describe concrete methods for the automated creation of delta, merge, and composite grammars; for the discovery of inconsistencies in extensions after a core grammar update; for inline rule refactoring; and for the conversion of EBNF grammars to the ANTLR syntax. Understanding this requires some basic knowledge about Eclipse, Xtext, ANTLR, and Xpand, which was provided in Chapter 2. For any further details, the Eclipse Application Programming Interface (API) [26] is a very good source explaining most of the classes and methods that come into play here.

4.1 Using BNFTools

All of the methods, excluding the EBNF to ANTLR conversion and the rule inlining, were implemented as pop-up menu features in Eclipse. These actions are available for any EBNF file (extension ".bnf"), but most methods require a certain grammar subtype, such as merge or delta grammars. The following figure (Figure 1) depicts an Eclipse environment with BNFTools installed. It shows a selected EBNF file and its context menu, including the features we introduced.

We are first going to present how to complete the four-stage composition process, as described in the previous chapter, starting from a core grammar and its extensions all in EBNF form. The first step is to convert each of the extensions to the delta notation via the **Generate delta EBNF from extension grammar** feature. The corresponding method needs to resolve the core grammar, which can be done either implicitly (if there is only one imported grammar) or by explicit specification in the corresponding import statement. Once all the extensions have been converted to the delta notation, the user may proceed with the creation of a merge grammar via the **Generate merge grammar from a core grammar** feature. Once a user issues this action on the core grammar, they will be prompted to select the delta grammars they wish to merge it with, and afterwards the new grammar is created. Finally, the user can complete the composition process by selecting the next pop-up menu feature, **Create a composite grammar from a merge grammar**, on the merge grammar. This feature requires no input and ouputs the core grammar, extended with the selected packages, the interaction between which (if any) was specified by the operators of the merge grammar.

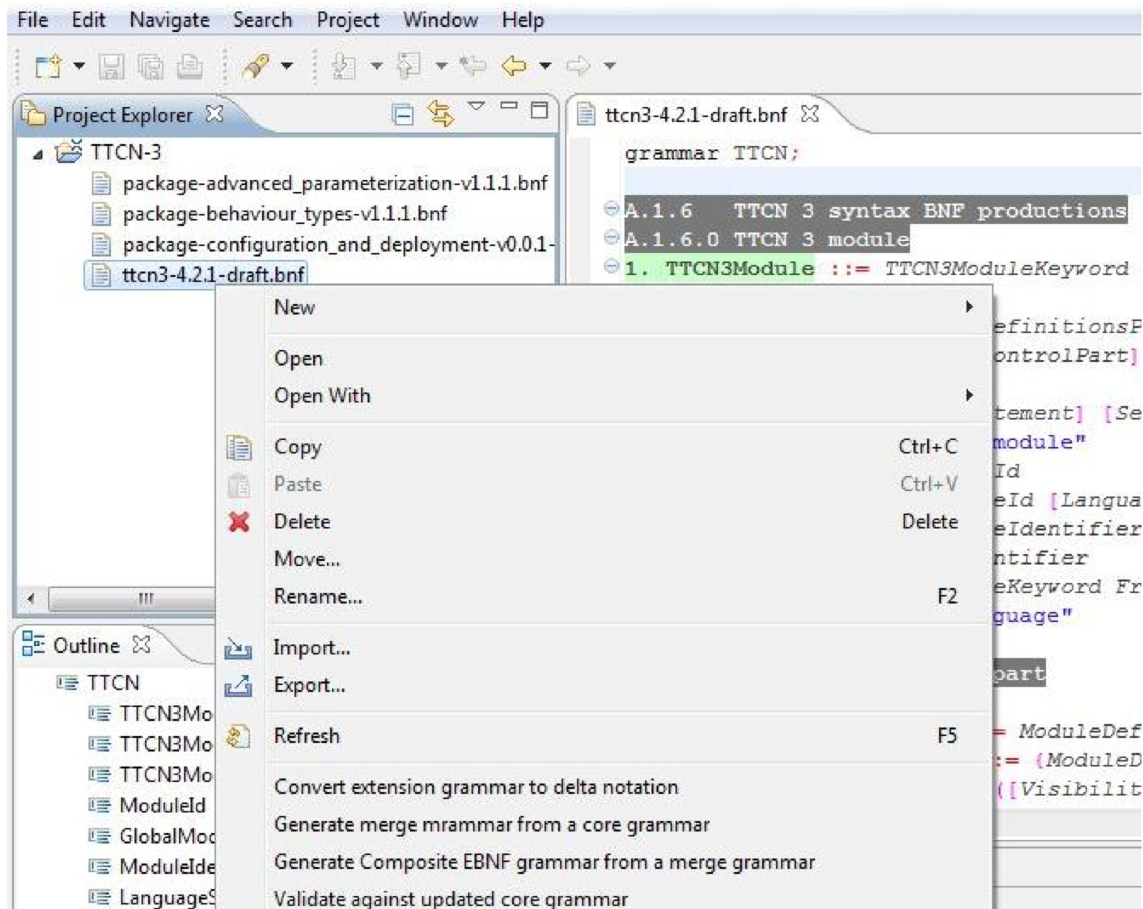

Figure 4.1: Eclipse with BNFTools

Another new feature handles the discovery of extension rules that are no longer valid after a core grammar update. This was again done with a pop-up action, called **Validate delta grammar against updated core grammar**, which has to be issued on a delta grammar. In this grammar the old and the new versions of the core grammar have to be specified as import statements with the keywords "/core" and "/update". The output is a plain text file in which all the inconsistencies caused by the transition to the new version are shown.

Rule Inlining was implemented as a **heavy validation** feature, that is a syntactical validation that is turned off by default. In order to use it, the user has to select **Validate** from the context menu of any EBNF grammar. The rules that can be inlined will then be highlighted and the automated inlining can then be called from the quickfix options.

4.2 Conversion to Delta Grammars

The first stage of merging grammars is converting the packages to delta grammars. We iterate over the imports of the selected grammar: if only one exists, we assume it is the core grammar. Otherwise we look for an import with the "/core" keyword. Afterwards, we obtain the parse tree of the selected grammar. Then, for every rule in the extension grammar we iterate over all the rules of the core grammar until we find one with the same name. If such a rule is found, we split it into one or more extension rules. Else, we simply we print it as is.

The *splitRule* method creates extension rules given two regular rules with the same name. It follows two main guidelines: there has to be an extension rule for every index at which the core rule is extended and there can be a maximum of one extension rule per index. The following steps are performed by the method:

1. Get a list *cLeaves* of all tokens of the core rule in the order they occur in the grammar. Initialize a corresponding counter *c* with zero.
2. Get a list *pLeaves* of all tokens of the extension rule in the order they occur in the grammar and initialize its counter *p* with zero.
3. Create an integer variable *offset* that keeps track of the number of tokens that have been added to the rule.
4. Create three boolean variables: *inExtension* checks if the last iteration included an extension token, *leftSide* is true if we are still to pass the left side of the core rule, and *leftSide2* is the same for the extension rule's left side.
5. Create two empty strings: *temp* to hold the extensions' left side and *temp2* for the right one.
6. Iterate while *p* is less than the size of *pLeaves*:
   a) If *leftSide* is true, add the serialised core token to *temp* and increment *c*. If the token is "::=" set *leftSide* to false. Continue.
   b) If *leftSide2* is true, increment *p*. If the token is "::=" set *leftSide2* to false. Continue.
   c) If *c* equals the size of *cLeaves*, add the serialized extension token to *temp2* and increment *p*. Continue.
   d) If either current token is a hidden one, increment the corresponding counter and continue.
   e) If *inExtension* is set to true and the tokens are different, add the current token to *temp2* and increment *p*. If the tokens are the same, set *inExtension* to false, create an extension line in the output file, and increment both counters. Continue.

f) If *inExtension* is false and the current tokens are equal, increment both counters and continue.

g) Initialise *temp2* with the empty string. Add the current extension token to it and set *inExtension* to true. Continue.

This algorithm only works if there are no tokens from the original rule that have been deleted or replaced (excluding hidden ones such as new lines). Its output is a "bnf" file with a name specified by the user that contains the package in delta form, unless an error occurred.

4.3 Creation of Merge Grammars

Merge grammars, the second stage in the grammar composition process, require several grammars as input, but their creation is relatively easy. At the start we ask the user to select the grammars they would like to merge with the core grammar. This part of the feature is error-tolerant: any of the selected resources that are invalid are ignored and a warning is produced. Valid selections need to be in delta form and import the core grammar.

We save all the grammars that abide to the above constraints in a global array, called *deltas* and create a label for each in another global array called *deltaLabels*. At the same time, we create the imports in the merge grammar — one for each package, including its label. Afterwards, we invoke the *handleExtensions* method for each rule in the core grammar, described below:

1. Create a linked list of abstract nodes called *rules* to contain all the rules that extend the current core rule.

2. Iterate over all rules in all the resources in the global array *deltas*. If any of the rules has the same name as the current rule, add it to *rules*.

3. If the list is empty, return. Else, create two new lines, both as comments: the first containing a **RuleCombinator** with the name of the current rule and the **and** operator, the second a copy of the rule itself. Then, iterate until there are no more elements in *rules*:

    a) Initialize two nodes *old* and *current* that are going to contain the extension rules from the previous, respectively current, iteration.

    b) Assign the result of the *selectFirstExtension* method (described below) to *current*. This method returns the extension rule with the lowest extension index.

    c) Create an integer *index* containing the index value of *current*'s grammar in *deltaLabels*.

d) If this is the first iteration, set *old* to *current*.

e) If *old* and *current* have different extension indices, or if this is the first iteration, create a new **HookCombinator** with the given rule name and the extension index of *current*. Add the label of *current*'s grammar from *deltaLabels* on a new line, followed by a comment containing the delta rule from that grammar with the same rule name and extension index.

f) Else, if *old* and *current* have the same extension point, only create a new label with *current*'s grammar and the comment with the delta rule.

g) In both cases, create a new comment, containing the extension rule.

h) Set *old* to *current* and continue.

The method *selectFirstExtensions* is a simple and straightforward method that expects a list of extension rules as an argument and selects the one with the lowest extension index from them. This element is then saved in a separate variable, deleted from the list, and the variable is returned.

4.4 Generation of a Composite EBNF Grammar

The generation of a composite EBNF grammar involves the transformation of a merge grammar into a complete EBNF grammar that contains the core grammar and its packages. This method proceeds as follows: first the imports in the merge grammar are resolved and saved in global variables. We do not need any user input here as merge grammars use the *grammarType* property of import statements to specify their role in the grammar. The main method of the feature has two cycles: one that calls *findExtensions* for every rule in the core grammar and one that copies all regular (that is, non-extending) rules from all the extension grammars into the target file.

The method *findExtensions* in the first cycle takes a core rule as argument and adds it to the output file either as is or extended. It iterates over all the elements in the merge grammar and searches for the three types of merge rules. If a **GlobalCombinator** is found, a global variable is set to it. If a **RuleCombinator** with the same name as the current core rule is found, its value is saved in the local variable called *ruleC*, used to pass it to other methods. Finally, every **HookCombinator** for the current rule that is discovered is added to a list. If, after going through the whole merge grammar, this list is empty, the rule is printed as is, i.e., there exist no extensions. Else, the method *handleExtensions* is called with arguments the current rule, the list of extensions and the *ruleC* variable (can be null). The result of the method is printed in the output file.

The method *handleExtensions* expects an EBNF rule, a list of its extensions as **HookCombinators**, and a string that either represents one of the composition operators or null.

The method returns a string representing the rule extended in the way specified by the strongest operator. The way it works is as follows:

1. Initialize two integer variables with zero: *newOffset* for the current hook position and *oldOffset* for the position from the previous iteration.
2. Initialize a new string *fullRule* with the left side of the current rule. This string will be extended to contain the whole extended rule.
3. Repeat until there are more than zero extensions:
    a) Get the extension with the lowest extension index and delete it from the list (via *selectFirstExtension* — same as in the previous subsection).
    b) Save its extension position in *newOffset*.
    c) Call *splitRule* with the current rule, *newOffset*, and *oldOffset* as arguments to get the part of the core rule between the old extension index and the current one. Append the result to *fullRule*.
    d) Call *resolveReferences* with the current HookCombinator and *ruleC* as arguments to get the extension portion of the rule at the current hook position. Append the result to *fullRule*.
    e) set *oldOffset* to *newOffset*.
4. Append the part of the core rule between the last extension index and the end of the rule by calling *splitRule* with the current rule, *newOffset*, and a max value integer as arguments.

The *splitRule* method returns a string containing the portion of the rule between two indeces: *startOffset* and *endOffset*. The method initializes a string *temp* and works by iterating over a list containing all the leaves of the rule. In each iteration there are four possible cases: if *startOffset* equals *endOffset*, return *temp* — we have copied all the necessary parts; else, if still on the left side of the rule — ignore the token and continue; else, if less than *startOffset* number of tokens have passed — continue; else add the current token to *temp* and increment *startOffset*.

The method *resolveReferences* takes a **HookCombinator** and a possibly empty string as arguments and returns a string containing the extensions described by the **HookCombinator** merged according to the most powerful composition operator. We are going to describe this method below, but first we are going to briefly introduce another related method: *resolveSingleReference*. This method takes as arguments a string representing a grammar label and a **HookCombinator** and returns the string describing the extension uniquely identified by the label and the **HookCombinator**. The way this is done is simple: the label is

resolved to the delta grammar it refers to by using the class variables we set at the start; the HookCombinator provides the name and extension point to search for in the grammar. This result is used in *resolveReferences* as described below:

1. Get the list with labels from the **HookCombinator** and save it the variable *labels*.
2. If the size of *labels* is one, there is only one extension at that index — resolve it (by calling *resolveSingleReferences*) and return the result.
3. Initialize a new string variable *combinator* to contain the combinator for this extension position. Set *combinator* to the operator described in the current **HookCombinator**. If no such operator exists and *ruleC* is non-null, set *combinator* to it. Else, if *global* is non-null, set *combinator* to it. Finally, if no operator is found, use **and** by default.
4. Continue depending on the *combinator* value:
   - "/and" Simply concatenate all the extensions in the order their labels occur by successively calling *resolveSingleReference* for each of them and appending the outputs to a string.
   - "/andr" The same as "and" but in the reverse order.
   - "/or" The same as "and" but we put each extension in parenthesis, with the "or" operator between every two. We also check every extension if it starts or ends with an invalid special symbol and if yes, we abandon the string created so far and create a new one in the same way we did for "and".
   - "/interleave" The target output consists of the three alternatives above, grouped in parenthesis. We start with the "'or"' one and if no problems are found, proceed with the "and" and "andr" ones.
5. Return the string from the previous step or an error if the operator was not recognized.

4.5 Other Features

Excluding *Validation after a Core Grammar update*, the implementation of these features was done differently from the features presented before and involves many technical details that we will describe briefly. For *Rule Validation* we extended the validation and quick-fix capabilities of the Xtext-provided BNFTools IDE, while for *BNF to ANTLR Conversion* we created an entirely new Xpand project that provided us with the environment for the implementation.

4.5.1 Validation after a Core Grammar Update

The only miscellaneous feature related to extension grammars is also the one that was implemented in the same way as the previous features. We start by validating the three necessary grammars: the selected extension should be in delta form; the core and the updated core grammar in EBNF. After resolving and validating the two core grammars, we save them in global variables. The main method is a simple iteration over all the rules in the delta grammar. The method *CheckForConsistency* is called for every delta rule and searches for changes in core grammar rules that are redefined by the package. It is presented below.

1. Initialize two variables — *coreRule* and *updatedRule* to contain the extended core rule and its updated version respectively.
2. Iterate over all the nodes in the updated grammar. Search for a node of type Rule with the same name as the current extension rule. If found, save it in *updatedRule*.
3. If *updatedRule* equals **null**, output an error statement in the file and return.
4. Else, repeat step 2 for the core grammar and *coreRule*.
5. If *coreRule* is still *null*, print an error statement about invalid core grammar and return.
6. Get the lists containing the leaf nodes of both core rules. Iterate over them and remove any leaf that represents a hidden token.
7. If the sizes of the new lists are not equal, print an error that there is a change in the updated version of the extended core rule.
8. Else, compare the serializations of both rules token for token. If any mismatch, print the same output as in the previous step and return.

4.5.2 Rule Inlining

Finding and inlining rules that are referenced only once is a straightforward algorithm and depends largely on the technical background of the implementation. The idea is simple: for every rule in the grammar, search through all the leaves of the grammar for **Rule References** with the same name as the rule. If exactly one is found, the rule can be inlined, else it cannot. The inlining itself consists of deleting the only token referencing the rule and inserting the serialization of the right side of the rule in its place. Afterwards, the rule is deleted and the inlining is complete. In our implementation the search for rules that can be inlined was implemented as a validation for Eclipse, while the inlining itself as a quickfix.

4.5.3 EBNF to ANTLR Conversion

The conversion to ANTLR notation is a simple model to text conversion. We traverse the Ecore-based parse tree that Xtext generates from the EBNF syntax in a depth-first manner. Most of the transformations are straightforward and easy to implement: cardinality operators replace the different kinds of parenthesis, single apostrophes replace the doubles, the colon replaces the operator for the start of the right side. A semicolon has to be printed after every rule.

The only non-straightforward change is the renaming of the **rules** and **rule references**: all terminal rules have to start with a small letter, and all the nonterminal rules with a capital letter. This was implemented by making a second walk of the tree. In the first pass we check the number of **rule references** in each **rule**: if none are found, the rule is stored in a global array of nonterminal rules, else it is ignored. In the second pass the actual grammar is printed. In addition to the modifications presented above, every **rule** and every **rule reference** is checked if it is contained in the array. If it is, then the rule name is printed in all capital letters, else in all small letters. The output grammar would likely still need to be modified before it is ready to be used in ANTLR, for example in order to solve left-recursion, but these changes are left to the user.

5 Summary and Outlook

In this chapter we are going to summarize the methodology and the results presented in the thesis. We are also going to take a look at the limitations of our approach and discuss ways to go around them by discussing both miscellaneous features that can be used for the maintenance of EBNF grammars in general and ideas for the improvement of the composition process in particular.

5.1 Summary

EBNF grammars have played a vital part in the creation of programming languages for a long time and are likely to preserve their importance in the future. In this thesis we have developed an approach to use EBNF grammars as a way of extending language syntax specifications, as well as a corresponding implementation. Unlike similar projects, we have only considered the initial stage of language creation: the syntax specification, but provide support for parser creation and beyond via ANTLR.

We have presented the composition of grammars as a four-stage process that allows any number of extensions to be merged at once. To that extent we have introduced two new domain-specific languages: the delta and the merge grammars. Delta grammars provide a concise way of describing EBNF extension grammars by differentiating between rules that redefine core grammar rules and rules native to the extension. Merge grammars support the merging of several extensions together: they highlight the possible interaction of the extensions with each other and provide a way to avoid ambiguities with the help of composition operators. We have specified four composition operators that regulate how extension rules from different grammars are connected together in the case they redefine the same core rule at the same rule position. We also present cases in which two of the operators lead to erroneous syntax, which was handled by using a default operator instead.

Besides the composition of grammars, we have also discussed several miscellaneous topics related to EBNF grammars. As a further support for language extensions we have developed a method that finds extension grammar inconsistencies caused by a core grammar update. We have also proposed a method for inlining rules of an EBNF grammars, which may ease the maintenance and increase the performance when working with large grammars at the cost of reduced readability. Finally, we have provided a way to convert EBNF grammars to the ANTLR notation, from which a lexer and a parser may be derived.

In addition to the theoretical basis, we have also presented several algorithms that can be used to implement the discussed approach. We have focused on facilitating the composition process by providing methods for the automated creation of delta, merge, and composite grammars. We have also shown algorithms that support grammar validation, rule inlining, and EBNF to ANTLR conversion. All of these algorithms have been implemented as an extension to the BNFTools plug-in for the Eclipse Platform.

5.2 Outlook

There are several ways in which EBNF grammars and their extensions can be further improved. On the one hand, there are many ways in which the maintenance of EBNF grammars in general can be made easier, such as methods for detecting left recursion, detecting shift-reduce and reduce-reduce conflicts, automated left-factorings, and so on. Such solutions would aid greatly in preparing the grammar for the work with parser-generators, such as ANTLR.

On the other hand, there are also several ways in which our approach to the composition of grammars can be improved, for example, by providing several ways to handle errors caused by the "or" and "interleave" operators. Furthermore, in the case of the "and" operator it may be useful to be able to specify a concrete order of the extensions, instead of just choosing between "as declared" or "reverse".

There is, however, one idea that naturally builds up on the composition process presented in this thesis. So far we have only considered extension grammars subject to one constraint: extension rules have to preserve all elements of the original rule and may only add new elements to it. We are going to explore four possible ways to handle rules that also replace or delete elements of a core rule. We are going to refer to such extension rules as *replacement rules*. We are going to use the following two rules as examples. In the core rule there is a literal "public" that is replaced in the extension grammar by a reference to the rule *Modifier*.

```
1 Core: Header ::= "public" "class" ID
2
3 Extension: Header ::= {Modifier}+ "class" ID
```

Listing 5.1: A Core Rule and a replacement Rule

The main problem we face when dealing with replacement rules is handling the deleted elements, especially when merging several extension grammars. Several possibilities exist - delete them, add the whole original rule as an alternative to the new rule, make the deleted elements optional or treat the deleted part as an extension. Each of these has its strengths and weaknesses which we will present below:

- Deletion — we use the replacement rule as is, consequently deleting elements from

the core rule. This is the most obvious and straightforward approach and it guarantees that the complete grammar will have the syntax meant by the extension. There are two problems however: no compatibility with the original syntax and no way to use multiple extension grammars together.

```
1  Header ::= {Modifier}+ "class" ID
```

Listing 5.2: Deleting the Replaced Tokens

- Alternative — the newly created rule consists of two (or more) alternatives — the core rule and the replacement rule(s). This guarantees backwards compatibility with the core grammar and is easy to extend to multiple grammars, but at the cost of an increased number of rules.

```
1  Header ::= HeaderA | HeaderB
2  HeaderA ::= "public" "class" ID
3  HeaderB ::= {Modifier}+ "class" ID
```

Listing 5.3: Preserving the Rule as an Alternative

- Optional Element — we add optional brackets to the replaced elements and then insert any extensions/replacements in the way we presented in this thesis (using composition operators and so on in the case of multiple extensions). This approach improves the performance, but needs further specification — for example, the replaced elements may be inserted before or after the new ones.

```
1  Header ::= {Modifier}+ ["public"] "class" ID
```

Listing 5.4: Preserving the Rule as an Alternative

- Treat as Extension — we can use the approach of this thesis and simply treat the deleted parts as extension to the core rule. This would be easy to implement and is still customizable, but it may create unexpected syntax because of the presence of elements that are supposed to be missing.

The four variants could be realized in a similar manner with distinct merge operators that guide the composition process for replacement rules.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] ANTLR. Antlr Parser Generator. `http://www.antlr.org/`.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the Algorithmic Language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[4] H. Behrens, M. Clay, and S. Efftinge. Xtext User Guide. `http://www.eclipse.org/Xtext/documentation/1_0_0/xtext.pdf`, 2010.

[5] W. Bilofsky. PDP-10 IMP72 Reference Manual Version 1.5, August 1972.

[6] J. Earley. Syntax Extension Using a Run Time Model. *International Journal of Parallel Programming*, 3(3):189–196, September 1974.

[7] ETSI. TTCN-3 Language Extensions: Behaviour Types, 2010. ES 202 785, V. 1.1.1.

[8] ETSI. TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing, 2010. ES 202 782, V. 1.1.1.

[9] ETSI. TTCN-3 Standard: Core Language, 2010. ES 201 873-1, V. 4.2.1.

[10] D. Frej. Antlr Documentation. `http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation`, September 2009.

[11] S. Gonzalez and W. De Meuter. Domain-Specific Language Definition Through Reflective Extensible Language Kernels, July 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, The Third Edition*. Addison-Wesley Professional, 2005. `http://java.sun.com/docs/books/jls/third_edition/html/syntax.html`.

[13] IETF. Augmented BNF for Syntax Specifications: ABNF, 1997. RFC 5234.

[14] E. T. Irons. Experience with an Extensible Language. *Commun. ACM*, 13(1):31–40, 1970.

[15] IEC ISO. Information Technology — Syntactic Metalanguage — Extended BNF, 1996.

[16] Jetbrains. Meta Programming System. http://www.jetbrains.com/mps/.

[17] Jetbrains. MPS User's Guide. http://confluence.jetbrains.net/display/MPSD1/MPS+User's+Guide+(one+page).

[18] J. Leffler. BNF Grammars for SQL-92, SQL-99 and SQL-2003. http://savage.net.au/SQL/index.html.

[19] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):480–499, 1997.

[20] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[21] J. Paakki. Attribute Grammar Paradigms — a High-Level Methodology in Language Implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[22] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[23] Eclipse Project. Eclipse Modeling — EMF. http://www.eclipse.org/modeling/emf/.

[24] Eclipse Project. MWE Official Documentation. http://wiki.eclipse.org/MWE_oaw4.3_doc.

[25] Eclipse Project. Xpand Eclipsipedia. http://wiki.eclipse.org/Xpand.

[26] Eclipse Project. Eclipse Application Programming Interface. http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/index.html, 2010.

[27] Eclipse Project. Eclipse Documentation. http://help.eclipse.org/helios/index.jsp, 2010.

[28] Eclipse Project. Eclipse Official Website. http://www.eclipse.org/, 2010.

[29] Eclipse Project. Xtext Official Website. http://www.eclipse.org/Xtext/, 2010.

[30] Kragen Sitaker. C BNF Grammar. http://lists.canonical.org/pipermail/kragen-hacks/1999-October/000201.html, 1999.

[31] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an Aspect-Oriented Extension to the C++ Programming Language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[32] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2009.

[33] M. van den Brand, P. Klint, and J. Vinju. The Syntax Definition Formalism. `http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.pdf`, 2007.

[34] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[35] B. Wegbreit. The ECL Programming System. In *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference*, pages 253–262, New York, NY, USA, 1971. ACM.

[36] E. Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute Grammar-Based Language Extensions for Java, 2007.

[37] D. Zingaro. Modern Extensible Languages, 2007.