UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Master's Thesis

# T□xt

## An approach to mix EMF based visual and textual editors

Submitted on:

September 17th, 2012

Submitted by:

Stefan Kuhn
Goethestr. 18
67251 Freinsheim, Germany
Phone: +49 (163) 87 48 2 05
E-mail: wedel@stefan-kuhn.eu

Supervised by:

Prof. Dr. Ulrich Hoffmann
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +49 (41 03) 80 48-41
E-mail: uh@fh-wedel.de

# T□xt

## An approach to mix EMF based visual and textual editors

Master's Thesis by Stefan Kuhn

This thesis presents an approach to enable mixed interchangeable textual and graphical editors on EMF models which still conform a context grammar. It unifies the concepts of Sentential Forms with structured, presentable tokens. Synthetic characters allow to resolve structured information from a single character. Alternative graphical editors can be created after the extended language was shipped.

Layout based on Björn Peemöller's, Stefan Roggensack's and Timo B. Hübel's template as well as LaTeX $2_\varepsilon$, KOMA-Script and BibTeX.

# Contents

# List of Figures

# 1 Introduction

Studies show that software developers spend more time understanding source code than writing it [KAM05]. Lowering the cognitive burden to understand the presented program sources is a key factor to increase productivity.

Presenting a program specific to the qualifications and domain of the reader helps to lower the entrance barrier. Domain specific languages help non-software developers to be integrated in the developing process and help them to understand and correct the program. Still not even a single domain specific representation is best for all tasks involving the same program [Moh93]. Embedded or internal domain specific languages have a low entry barrier, but lack the ability for visual editing and domain specific feedback. Because they are embedded in a host language, internal domain specific languages have to follow their syntax constraints. Internal domain specific languages are source code, which is often repelling for non-software developers. External domain specific languages are flexible in notation and user guidance, but development and maintenance costs are higher. Additional tools or an additional tool chain is required and changes in the language are much more effort. External domain specific languages are integrated in the runtime either by using their result as a configuration for an interpreter or by generating source code. Interpreting the word of a language completely decouples the external domain specific language from the source code. With the generative approach, generated source code from the domain specific description normally coexist side by side with hand written code. External languages are regularly not general purpose languages, otherwise their complexity goes towards the target language making their development expensive and the target language obsolete. Coexistence of generated and hand written code is not consensual, because source code generation is unidirectional and in most cases overwrites previously generated code. Integration with hand written code must be designed carefully because incremental development is hindered and synchronicity of the modeled and the source code state is not maintained. No fluent transition from internal to external domain specific languages is possible.

Modern integrated development environments (IDEs) are designed to maximize the productivity of its user. Thus, it leverages the plain source code representation of the source code by processing it and providing a comprehensive view on the program description. They provide navigation and refactoring based on computed meta data, allow the user to customize the view to a certain extent and manipulate the source code by predefined actions. The source code is the central description of the program, but visual viewing, refactoring and processing of the textual input is usually done on an abstract data structure gained by parsing the textual input. The parsing is mainly based on a context free grammar for a textual language.

Modeling tools for visual languages on the other hand are strongly based on the model view controller pattern and thus separate the notation for a language with their abstract description. The well known Unified Modeling Language 2 (UML2) uses a model to formally describe the abstract syntax of a language and constraints to describe its static semantics. These models of languages form the common denominator for tools and frameworks for working with the described language. General use cases are model transformation, model presentation in a textual or graphical manner, model validation, model persistence and so on. These models integrate tools which are able to create, present and process instances of a described language; they form the common ground for component oriented language workbenches.

Several tools [1] already bridge the gap between the description of a language by a context free grammar and by models. They require a full textual representation of a model element and its content does not allow the user to select alternative

[1]XText, Textual Editing Framework, EMFText

textual presentations of semantic equivalent elements. Projectional editors like Meta Programming Systems offer form based textual views on data structures, whose design approach is predestined to show just parts of the model or to present certain model elements differently. Projectional editors enforce a special user guidance, which is more restrictive to free textual editing. Additionally, they require continuous structural correctness while editing, which enforces the user to make edits in steps which are at least structurally correct. This contradicts well-known editing behavior, where users create several incorrect intermediate states of a document to reach the desired, correct state. Abandoning this editing behavior restricts the user in their familiar work flow. It requires them to not only learn the model manipulating tool, but also to learn and to switch to another editing paradigm.

This thesis started from the basic idea to leverage internal domain specific languages with domain specific notation and to make domain specific manipulation facilities interchangeable. This requires a mix of interchangeable textual and graphical notations and integration of the editor in the word as if it was part of it. So the data in place of the editor has to be part of a word. This word should be in language described by a context free grammar.

## 1.1 Research Objective

The central questions of this thesis are:

- How are different, interchangeable textual representations of the semantically same abstract language element in general possible?
- How can graphical editors be integrated into a text, so that the text is still a word of the language described by a context free grammar?
- Which granularity of integration is possible if different editors are unknown at design time of the context free grammar?

This thesis uses an abstract notion of graphical editors in which the type of editor, its position in the character stream and a proper data source is sufficient to facilitate their deployment.

## 1.2 Outline

The paper is structured as follows: chapter 2 provides an overview of several definitions and existing language or editor related solutions, chapter 3 outlines the research methodology, chapter 4 presents the results, chapter 5 discusses the findings and identified limitations and chapter 6 summarizes the paper.

# 2 Related Work

This chapter starts with the 2.1 Definitions, where a common understanding of terms are determined and, if necessary, specified. In the next chapter, general non-Metamodeling related approaches of Language Workbenches, language definition and editors are presented. In the final section 2.3 of this chapter, Metamodels are explained and Metamodel related frameworks are presented.

## 2.1 Definitions

This section contains definitions in alphabetical order.

**Abstract Syntax** Meta Model elements and their relation are the abstract syntax of a language. [SV05]

**Abstract Syntax Tree (AST)** The abstract syntax tree represents the hierarchical syntactic structure of a language. Abstract syntax trees resemble Parse Trees to an extent. In the Parse Tree, interior nodes represent nonterminals including "helpers" of one sort of another. In the syntax tree, these helpers are typically dropped [ALSU06].

**Alphabet** "An alphabet is any finite set of symbols." Examples of symbols are letters and digits [ALSU06].

**Ambiguity** A grammar which can generate more than one Parse Tree for a given string of terminals is said to be ambiguous [ALSU06].

**Concrete Syntax** In contrast to the Abstract Syntax, the Concrete Syntax defines the actual representation of the language [SV05].

**Constraint Satisfaction Problem (CSP) in regard to Parsing** Besides answering the question if a word is part of a language, a practical reason for parsing is to obtain the structure of a word to help processing or translating it further. To parse a word according to a grammar means to reconstruct the production tree that indicates how the given word can be produced from the given grammar. This is a specific Constraint Satisfaction Problem (CSP): To consume all tokens on the input stream constrained by the grammar rules. It is possible to build the Parse Tree by the path taken, or the sequence of rule applications by the parser. Top-down parsers identify the production rules in post-order, bottom-up parsers tend to identify them in postfix order. This is called Linearization of the Parse Tree [Gru10].

**Context Free Grammar (CFG)** A context-free grammar is described by a quadruple, consisting of terminals, nonterminals, a start symbol and productions.
$G = (T, V, S, P)$

1. *Terminals* are the basic symbols from which words are formed. They are created by the lexical analyzer.
2. *Nonterminals* are syntactic variables that denote sets of words. The sets of words denoted by nonterminals help to define the language generated by the

grammar. Nonterminals impose a hierarchical structure on the language that is essential to syntax analysis and translation.

3. One nonterminal in a grammar is distinguished as the *start symbol* which is an element of $V$

4. The *productions* of a grammar specify how terminals and nonterminals can be combined to form words. Each production consists of:

   a) A nonterminal called left side of the production; this production defines some of the words denoted by the left side.

   b) The symbol $\rightarrow$ or `::=`

   c) The *right side* consisting of zero or more terminals and nonterminals. The components of the *body* describe one way in which words of the nonterminal at the head can be constructed.

$P$ is a finite relation from $V \rightarrow (V \bigcup T)^*$, where $^*$ is the Kleene star. Members of $P$ are productions of the form $u \rightarrow v$ [ALSU06].

**Domain** A domain is a delimited area of knowledge or interest [SV05].

**Domain Specific Language** A Domain Specific Language is a Programming Language for a domain. It consists of abstract and concrete syntax, static semantics as well as a clearly defined semantic of the language elements [SV05].

**Extended Backus-Naur Form (EBNF)** The Backus-Naur Form (BNF) is a notation to describe context-free grammars. It is based on [Bac59] and defines `:=` and `|` as metaliguistic connectives and characters enclosed by `<` and `>` as metalinguistic variables.

```
<nonterminal> ::= sequence
```

The sequence consists of one or more nonterminals or terminals and may contain a vertical bar to indicate a choice. The whole sequence replaces the symbol on the right. The extended Backus-Naur Form (EBNF) is a family of BNF dialects with the same power as BNF, but with improved readability. Common extensions are grouping, the Kleene star denoting a zero or more multiplicity, the Kleene cross denoting a multiplicity of one or more and an option denoting a multiplicity of zero or one. [Gru10]

**Kleene Star** The (Kleene) closure of a language $L$, denoted $L^*$, is the set of strings received by concatenating $L$ zero or more times. [ALSU06]

**Language** A language is any countable set of words over some fixed alphabet. The set of all context-free languages is identical to the set of languages that is accepted by pushdown automaton. [ALSU06]

**Language Element** The term *Language Element* is used in this thesis to denote an `EObject` instance of a Metaclass of the language Metamodel. This is transferable to other Metamodels, for example, Notation element, etc.

**Language Workbench** A Language Workbench is a generic tool to build software around a set of domain specific languages. The key concept is that editing is not done on text files but on the *abstract* representation of a program [Fow05]. This enables the following characteristics:

- languages are closed under composition.
- users edit this abstract structure using projectional editors.
- a language has three main parts: abstract structure definition, editor(s) and optional generators or interpreters.

In practice, the term Language Workbench is often used for frameworks that support language creation to a greater extent than "compiler compiler" or parser generators. The bidirectionality of abstract structure and concrete editors is especially often "eased".

**Lexeme** A lexeme is a sequence of characters on the character stream that matches a specific token pattern and is identified by the lexical analyzer as an instance of that token [ALSU06].

**Metaclass** If the model class $c$ is an instance of the class $M$, $M$ is the Metaclass of $c$.

**Meta Metamodel** A Meta Metamodel is a model that describes Meta Models. Meta Metamodels, like Ecore and Meta Object Facility, are self descriptive: their metaclasses can be used to describe themselfs, so they are their own Meta Model [SBPM09].

**Meta Model** The structure of a language is captured in its Metamodel. A Metamodel *is a model* that provides the basis for constructing other, so called, instance models [Gro09]. The Meta Model defines the Abstract Syntax and optionally the Static Semantics of a language [SV05].

**Model** A Model or instance Model is an instance of a Metamodel.

**Lexer** A Lexer, also denoted as lexical analyzer, allows a parser to treat multi-character constructs, like identifiers, as units called tokens during syntax analysis. [ALSU06]

**Parsing** "Parsing is the process of structuring a linear representation in accordance with a given grammar." [Gru10].

**Parse Forest** A set of Parse Trees. If duplicated subtrees are combined, the resulting structure is called a parse graph. [Gru10]

**Parse Tree** Given a context-free grammar, a Parse Tree, according to the grammar, is a tree with these properties [ALSU06]:

1. the start symbol is the roots label.
2. The leafs are labeled by a terminal or by $\epsilon$.
3. Interior nodes are labeled by nonterminals.

4. If $A$ is a nonterminal, which labels an interior node and $X_1, X_2, ..., X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1X_2...X_n$. So $X_1, X_2, ..., X_n$ stand for a symbol each of which is either a terminal or a nonterminal. If $A \rightarrow c$ is a production, then a node labeled $A$ may have a single child labeled $\epsilon$ as a special case.

**S-Attributed Grammar** An Attributed Grammar formally defines a way to attach attributes to tree nodes created in regards to production rules of a formal grammar. If the evaluation rule of an attribute value only depends on its children, it's called a synthesized attribute. An S-Attributed Grammar is an Attributed Grammar which is only based on *synthesized* attributes. The evaluation of S-attributed grammars can be incorporated in top-down, as well as in bottom-up parsing. [Gru10]

**Sentential Forms** The intermediate form for creating sentences from a formal grammar are called sentential forms. [Gru10]

**Stable versus Unstable Models** This thesis uses these terms to distinguish the stability of references to model elements. Stability is no intrinsic characteristic of the model, but a contract of all model manipulation parties. Models which elements are *updated* if edited are called "Stable Models". If the model, or parts of it are *replaced* by equal model elements, the model is *unstable.* The worst case of unstable models is the creation of an completely new equal model if an editor commits an empty update.

**Static Semantics** The Static Semantics defines well-formedness constraints. [SV05]

**Subtree** Subtree denotes a branch of a tree.

**Syntax & Semantic** The syntax of a language describes its proper form, while the semantics of the language defines what it means. [ALSU06]

**Syntax Directed Editor or Projectional Editor** A Projectional Editor allows the user to directly edit the AST representation of the code or the structure of the document [jet12]. In general, syntax-directed editors allow partial presentations of the document and offer support for the interpretation extra state, for example, as built-in tree views on the document structure [Sch04].

**Syntax Recognizing Editor or Syntax Aware Editor** A syntax-recognizing text editor is one in which the user provides text and the system infers the syntactic structure by analysis [BGVDV92].

**Tokens & Terminals** A token consists of a name and an attribute value. The names are abstract symbols used by the parser for syntax analysis. These token names are often called terminals. The attribute value contains additional information. [ALSU06]

**Word** A word over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms *sentence*, *word* and *string* are used as synonyms. [ALSU06]

## 2.2 General Approaches

This section provides a selected overview of current language and editing approaches which are not Metamodel based. This includes Language Workbenches, tools to develop Domain Specific Languages, projectional and textual editors.

### 2.2.1 ANTLR

ANTLR [1], "ANother Tool for Language Recognition", is an open source parser generator framework for LL(*) parsers with selective backtracking. It allows the creation of Java parsers and lexers from a grammar file. The grammar file contains among other things an EBNF grammar with actions [Par07].

[1] `http://www.antlr.org/`

### 2.2.2 Barista

Barista [2] is an editor framework for user interfaces that offers alternative structured visual representations on an internal abstract syntax tree. It uses Citrus [3] as a user interface toolkit [KM05] and adds data structures and a textual interaction technique to it. It mimics the interaction technique of conventional text editors to overcome a central usability problem of previously structured code editors by fluidly changing between structured and unstructured text. It is based on the model view controller pattern [GHJV95] and the presented editors structure depends on the models structure. The model in Barista consists of *Structures* and *Tokens*. Structures are tree nodes and tokens the leafs. A token is assigned to a property of a Structure and is contained in it. For each Structure, a grammar is inferred or provided and for each token, a regular expression is specified that defines its legal strings and white space. The general idea of Barista's textual interaction technique is to unparse, retokenize and reparse a token if its value is not part of the language described by the regular expression. Barista uses a variant of [WG98] incremental parsing Algorithm to reuse the abstract syntax tree's structure [KM06].

### 2.2.3 Harmonia

The Harmonia Research Project [4] is a framework for constructing interactive language-based programming tools. It emerged from two earlier prototypes: *PAN* and *ENSEMBLE*. Harmonia's goal is to facilitate building interactive tools, so the provided services are *incremental* and the internal program representation maintained by the framework is *updated* as a program is manipulated. Harmonia provides syntactic and semantic analysis services [Bos01] and can be used to augment text editors with language aware editing and navigation functionality. Harmonia adopts from ENSEMBLE the incremental Lexer algorithm [WG97] and the incremental Parser algorithm [WG98]. Incremental parsing utilizes persistent Parse Trees and the parse input consists of both terminal and *nonterminal* symbols [WG98]. Ensemble also has a graphical presentation formalism based on tree transformations [Mav97]. This transformed graphical notation is not editable.

[2]`http://www.cs.cmu.edu/~NatProg/barista.html`
[3]`http://www.cs.cmu.edu/~NatProg/citrus.html`
[4]`http://harmonia.cs.berkeley.edu/harmonia/`

### 2.2.4 Integrated Development Environments

Integrated Development Environments (IDEs) like Eclipse, IntelliJ IDEA, Netbeans and Visual Studio parse the source code as it is typed; this is called Background Parsing. They use the parsed abstract syntax tree to provide editor services like outlining, refactoring, error marking and content completion. These are syntax-recognizing editors. LavaPE [5] is an example of an IDE with a syntax directed editor.

### 2.2.5 JetBrains Meta Programming System

JetBrains Meta Programming System [6] is an Java based open source language workbench, which allows to define the abstract syntax of a language, a projectional editor and model transformation. In the Meta Programming System, the abstract syntax of a *Language* is defined using *Concepts*. Model nodes are defined by their concepts [MPS12]. The terminology in the Meta Programming System abstract syntax is Language, Solution and Concept, where Language is similar to a Metamodel, Solution to a Model and Concept to a Metaclass. It provides a projectional editor that directly edits the abstract language instance directly. The editor focuses on text presentation, but it is not free, but form oriented text. At the time of writing, tables are the only editable graphical notation. Also, a Concept is either presented textually *or* as a table. A Solution can then be transformed by textual model to model transformations to one of Meta Programming Systems several base languages, which provide the semantics of the language constructs. As base languages, there is currently C, Java, XML or plain text available. Because the textual notations in the Meta Programming System are not free text, it requires the user to change his “editing habits” [Vol11].

### 2.2.6 Proxima

Proxima [7] is a generic structure editor written in Haskell for a range of structured documents. It allows free text editing as well as structure editing. Proxima maintains a bidirectional mapping between the document structure and its presentation, by

[5] `http://lavape.sourceforge.net/`
[6] `http://www.jetbrains.com/mps/`
[7] `http://www.cs.uu.nl/wiki/bin/view/Proxima`

a multiplicity of bidirectional mappings between the seven layers of Proxima's architecture. Proxima does not use conventional parsing techniques, because the presentation in not restricted to text, but also contains graphical elements. Graphical presentation can not be edited directly at the presentation level [SS08]. Proxima uses *structural tokens* for presentation that are not strings and treats them specially [Sch04].

### 2.2.7 Spoofax

Spoofax [8] is an Eclipse based framework to develop textual domain specific languages with IDE support [KV10]. Spoofax uses the Syntax Definition Format [9] (SDF2) to specify the syntax [KV10]. It combines the specification of concrete and abstract syntax into a single grammar. SDF defines context free grammar like EBNF [HHKR07], but it is among other things modular and together with scannerless generalized parser like [Vis97] and declarative disambiguation rules, it is freely composeable. Spoofax uses Stratego [10], a term rewriting or transformation tool [BKVV08], to process the abstract syntax trees. The abstract representation is, in contrast to EMF, based on trees.

[8] `http://strategoxt.org/Spoofax`
[9] `http://syntax-definition.org/`
[10] `http://strategoxt.org`

## 2.3 Metamodel Related Approaches

Metamodel related approaches describe the abstract language structure with a model for *the language*, which conforms to a Metamodel *for languages*. When the language model is used, an instance model of it is created. For example, the Unified Modeling Language [11] 2 (UML2) from the Object Management Group [12] formally defines, besides the notation, the structure of UML2 models in their Superstructure [Obj10b]. The Superstructure's structure is described using UML2 Infrastructure [Obj10a], which is also called Meta-Object Facility (MOF). The relationship of MOF to UML2 is similar to the relationship of BNF to a programming language's grammar.
One advantage of Metamodeling is that formal models or languages are created and that tools for supporting Metamodels can be built, which integrate if they use a common meta Metamodel.

This section first explains the relation of MOF with Ecore, the Metamodel of Eclipse Modeling Framework (EMF), and their interchange. Afterwards, EMF is explained in general, followed by a description of projectional editors based on EMF and finally, EMF related textual editing is addressed.

**MOF versus Ecore** Meta-Object Facility (MOF) is a modeling concept comparable to Ecore. It is defined by the Object Management Group [OMG] to define UML2. Essential Meta-Object Facility (EMOF) is the lightweight core of MOF that closely resembles Ecore. EMF can read and write EMOF's serialization format [Gro09].

**XML Metadata Interchange (XMI)** XMI is the standard serialization format of EMF models in XML. The "XMI format (...) could technically be considered a concrete syntax, although it's sometimes called a serialization syntax." [Gro09].

### 2.3.1 Eclipse Modeling Framework (EMF)

"EMF is a Java framework and code generation facility for building tools and other applications based on a structured (data) model" [EMF]. It does not depend on Eclipse. EMF unifies a subset of Java, XML and a subset of UML by models, which

[11]`http://www.uml.org/`
[12]`http://www.omg.org/`

are definable using XML Schema, annotated Java interfaces or UML modeling tools. EMF bridges the gap between modelers and Java programmers by relating modeling concepts to a simple Java representation of these concepts [SBPM09].

EMF's runtime framework[13] allows data to be validated, persisted and edited in an user interface editor. Change recording and notification as well as meta data by reflection is available. EMF is the foundation for fine-grained interoperability and data sharing among tools, for example, Model Transaction [14], graphical editors like the Graphical Modeling Framework GMF [15], database persistence [16], model transformation and so on.

EMF also supports saving models and loads them from persistent resources. A specialized XML serialization format, called XML Metadata Interchange (XMI), is available as a default serialization. References can be persisted and they can be across multiple resources with load on demand, proxy resolution and unloading or proxy creation support. References can be bidirectional in EMF with referential integrity maintained. [SBPM09]

### Ecore Meta Metamodel

EMF uses a model to describe other EMF models. This model is called Ecore and is the center of the EMF world. A simplified version of the Ecore Metamodel which shows the central available types is presented in Figure 2.1. Ecore is itself an EMF model and can be described by itself similar to the possibility of describing the syntax of EBNF in EBNF. This makes Ecore its own Metamodel, thus Ecore is a meta-Metamodel. The concepts of Metamodels is simple: a model which describes a model is a Metamodel. A model which describes a Metamodel is a meta-Metamodel. [SBPM09] A Metamodel describes the abstract syntax of a language [Gro09] and adds validation rules to the static semantics of a language [SV05]

Ecore supports several high level concepts which are not directly available in Java, like multiple inheritance, bidirectional relationships and containment relationships as

[13]The runtime framework distinguishes the part of EMF's framework from the one which is necessary for code generation.
[14]`http://www.eclipse.org/modeling/emf/?project=transaction`
[15]`http://www.eclipse.org/modeling/gmp/?project=gmf-runtime`
[16]`http://wiki.eclipse.org/Teneo`

a specialization of bidirectional relationships. Cycles in containment relations are forbidden and an `EObject` might just be contained in exactly one other `EObject`. Containment relations form trees within models. The roots of these trees are model elements without an container `EObject` and the leafs are `EObjects` without outgoing containment relations.

Figure 2.1 shows a subset of Ecore. The central elements of Ecore are:

- *EObject* the common supertype of all Metamodel objects *and* instance objects. An `EObject` is similar to an Object in Java.
- An *EPackage* is the root of a model and contains `EClassifiers` and `subpackages`.
- *EClass* models classes. A class references a number of other classes as its `supertypes`. They are identified by name and contain a number of attributes and references which both are structural features.
- *EStructuralFeature* is the supertype of attributes and references, it aggregates attributes like multiplicity.
- *EAttributes* have a type and are identified by name. They model the components of an object's data.
- *EDataType* represents a single type which must not impose structure. They are directly related to Java types.
- *EClass* and *EData* types are both *EClassifiers.*
- *EReferences* model one end of an association. They are identified by name and hold a type of the referable object. If the association is bidirectional, the *eOpposite* is assigned to its inverse reference. [SBPM09]
- Ecore classes hold and *eAnnotation* reference. *Annotations* store information which were not considered fundamental enough to explicitly support them in Ecore's Metamodel, for example, documentation, OCL validation, XML serialization parameters. [Gro09]
- In addition to structural features, Ecore can also model the interfaces of behavioral features.

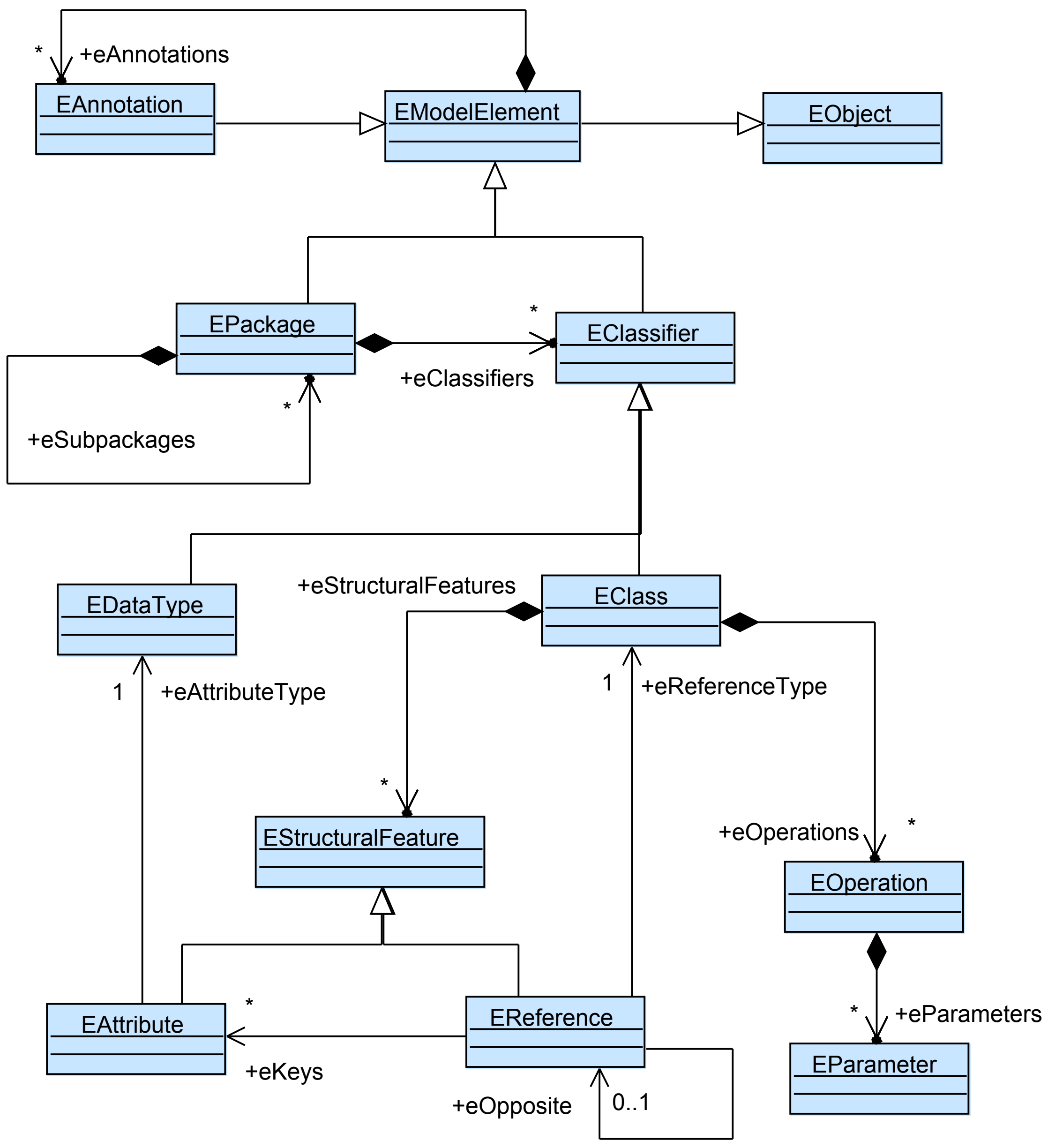

Figure 2.1: Simplified Ecore Metamodel

**Generator model** The generator model is the model that provides access to the data needed to generate the Ecore model by *decorating* it. This separation has the advantage that the Ecore model can remain pure without code generation dependent information with the disadvantage that it might get out of sync [SBPM09].

### EMF Runtime

EMFs runtime framework provides the basis for manipulation, change notification and serialization of EMF objects in general. [SBPM09] Change notification is based on the observer pattern [GHJV95], which are called adapters in EMF, because they frequently extend behavior of `EObjects` and adapt them. The behavior of adapters must be described in Java; EMF only declares behavior but does not specify it. All `EObjects` are observable, but also other EMF classes such as `Resource`s are.

In the following, just the persistence part of EMF's runtime is described. *EMF.Edit*, which is also part of it, is explained separately in the Projectional Editors section. XText [XTe12a] and EMFText [EMF12] integrate themselves in EMF as `Resource`, which is part of EMFs persistence concept. `Resource`, and `ResourceSet` are fundamental interfaces of EMF's persistence framework.

**Cross Resource References** To reference other persisted objects, EMF uses proxies. A proxy is an uninitialized instance of the target class, which is identified by an URI. "A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource." [BLFM05] A URI consists of five parts: scheme, authority, path, query and fragment, whereby query is not considered for this thesis. The basic format is:
`schema://authority/path#fragment`
The schema defines the interpretation of the following part. Common schemata for EMF are files, in an eclipse environment platform and http. Authority and path together identify an resource; the fragment identifies a part of a resource or, in the case of EMF, an `EObject`.
For example:
`http://www.xtext.my/MiniJava#//@class.1`
which identifies the the second element stored in the containment reference `class` of the resource `http://www.xtext.my/MiniJava`.

**Resource** A `Resource` is a basic unit of persistence. It is a container for one or more objects which are persisted together with their children. They can be unloaded, in which case contained referred objects are converted into proxies. A `Resource` is identified by a URI with schema, authority and path. A resource is also responsible for creating valid URI fragments for `EObjects` it contains. By default, a resource returns hierarchy based XPath like fragments. Other options are intrinsic IDs and extrinsic IDs. Intrinsic IDs are IDs stored directly on the model object, whereas extrinsic IDs are stored externally. Extrinsic IDs are useful when the modeled object's state is not sufficient to constitute an ID for a given resource. XML resources manage extrinsic IDs and offer support for universally unique identifiers (UUIDs). It is possible to attach adapters to a resource, so it is, for example, possible to observe every change of a contained object and create a `ChangeModel` by using EMF's `ChangeRecorder`. This enables automatic undo support for commands. [SBPM09].

**ResourceSet** A `ResourceSet` acts as a container for `Resources` and its main responsibility is to support references between objects in different resources.

### 2.3.2 Projectional Editors

This section describes two projectional editors for EMF models, EMF.Edit and the Graphical Modeling Framework (GMF). EMF.Edit is a framework for tree and table editors, whereas GMF is a full graphical editor.

#### EMF.Edit

With EMF.Edit it is possible to build editors for models, which display and edit (i.e., `Copy, drag-and-drop`, etc.) model instances with unlimited undo and redo. EMF.Edit is the controller in the model view controller pattern which uses JFace, a SWT based user interface toolkit, viewers by default. EMF.Edit and GMF controllers are in between view and model, so the view has no direct connection to the model. EMF.Edit supports object modification based on the Command pattern [GHJV95] and includes a number of common commands. A command offers `execute, undo, redo` and `canExecute` to check if all constraints are satisfied. EMF.Edit includes a set of generic commands based on the reflective API like `Set, Add, Remove, Move, Replace` and `Copy`, as well as higher level commands like `CreateChild, DeleteChild, CutToClipboard, CopyToClipboard, PasteFromClipboard` and `DragAndDrop`. Controllers are called `ItemProviders` in EMF.Edit, they offer to navigate their structure, modify their content, to retrieve labels, to forward notifications to the viewer as well as acting as a command factory. Controllers are in general associated with an `EObject`, but do not need to be, in order to let the editor display a structure different from the models structure. The central unit of EMF.Edit is the `EditingDomain` which encapsulates a `ResourceSet`, adds a command stack to support undos and redos as well as provides a controller factory [SBPM09]. EMF.Edit also provides a reflection based generic editor for *any* Metamodel instance.

### Graphical Modeling Framework (GMF)

GMF is a framework for creating graphical eclipse editors that are based on an EMF model. It's a controller framework using EMF models for persistence and Draw2D as view. GMF is based on the Graphical Editing Framework (GEF), and unifies GEF and EMF commands and extensively uses Eclipse extension points. Furthermore, it adds a Notation Model, which persists layout information independent from the language Metamodel and model. The concept of a Notation Model and how it is integrated in GMF is explained in the following paragraphs. The main purpose of GMF, the controller, is skipped because the much simpler controllers of EMF.Edit are sufficient in the context of this thesis. To bridge the gap, it's important to note that GMF creates a controller for each `View`, that is the basic model element of the Notation Model. Creating more than one `View`, e.g., by the view service factory, is a common practice in GMF to separate the presentation structure from model structure. Which `View` should be created is hinted by the Extensible Type Registry.

**Notation Model** GMF's Notation Model manages and persists the state of the diagram or `View` separated from the language model. It is completely domain model independent and thus can be handled generically by GMF's engine to provide common features. It manages the diagram element, position and style attributes. [Gro09] The base type of the Notation Model is `View`, which:

- holds a reference to the language model element it represents
- acts as the super type for `Node, Edge` and the Notation Models root: `Diagram`

- contains child nodes and references to edges
- has attributes like `visible` and `type`

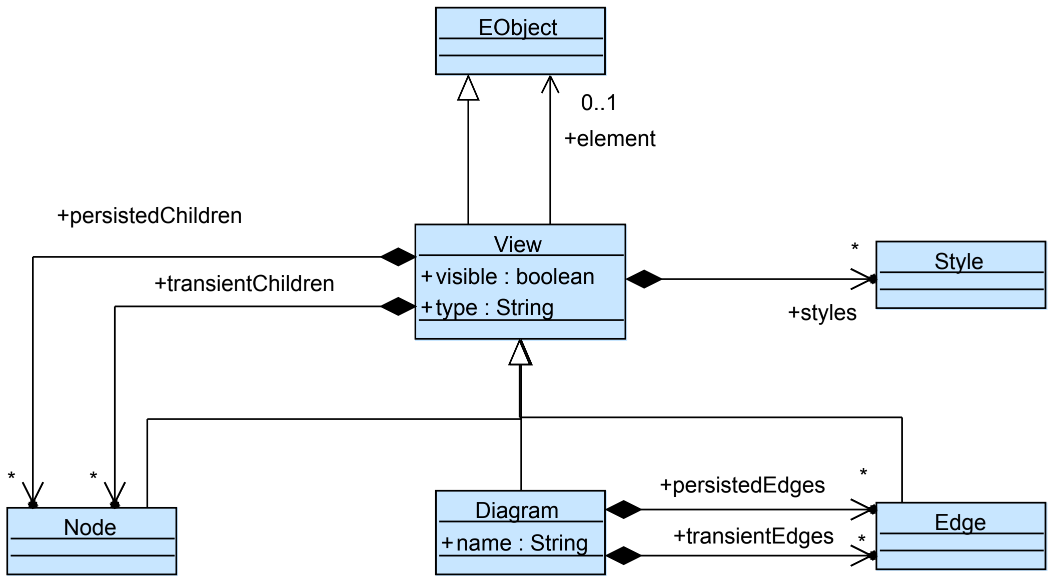

Figure 2.2: Simplified GMF Notation Metamodel

Part of the design considerations for the Notation Metamodel were [GMF]:

- The Notation Metamodel was designed to minimize merging conflicts through the separation of style into granular properties. They can be merged independently.
- The style hierarchy has been designed to be extendable; this allows individual extensions and enhancements in the future.

**View Service** The View Service is a factory for constructing new notation `View` elements also based on [GMF]:

- the corresponding language element (EObject)
- the type determined by the Extensible Type Registry.
- the container `View`

**Extensible Type Registry** "The Extensible Type Registry provides a way for GMF clients to define an application-specific classification system based on, but alternative to, the metaclasses defined by an Ecore Metamodel" [GMF]. This means that it is possible to introduce specialized GMF internal types, e.g., on the state of the referenced `EObject` or the state of another `EObject` referencing to the current one.

### 2.3.3 Textual Editing

This section describes how texts are integrated in EMF. At the beginning, the relationship between context free grammars and MOF based Metamodels is described. In the following, Xtext's grammar and architecture are described, based on the Xtext manual [XTe12a]. EMFText [EMF12] is besides Xtext the other well documented, still active developed textual framework for EMF. In contrast to Xtext, it allows the user to automatically derive a textual notation from a Metamodel and layout information in grammar rules. From the abstract point of view of this thesis, these are similar except that EMFText's documentation does not indicate how isomorphism from EClass to grammar rule can be circumvented. The relevance of this is discussed in section 4.1.1. This section closes with the editing strategy of Textual Editing Framework (TEF).

#### Relation of Metamodels and Context Free Grammars

To connect the textual languages, which are typically denoted by context free grammars and modeling languages, which are described by Metamodels, a mapping between the concepts of BNF and Ecore has to be established. Because Ecore is more expressive than BNF, it is possible to map BNF to Ecore, but just a subset of Ecore to BNF.

Algorithms for mapping a subset of MOF to BNF and BNF to MOF are described in [APCS03]. Note that [APCS03] uses a subset of MOF which is also a subset of Ecore and uses enumerations for fixed string literals (keywords), which is an optional specialization of the presented attribute mapping. Also, [APCS03] maintains an attribute order by generating lexical ordered structural feature names. Whereby an attribute order is crucial, this is not necessary regarding Ecore. EMF implements unordered multiple values as lists, which are ordered. This implementation characteristic is documented [SBPM09]. [APCS03] mentions that it's possible to create "invalid" models of the grammar Metamodel, which requires additional verification with the target context free grammar. This topic is discussed later in section 2.3.3.

The general idea of context free grammars to Metamodels mappings is:

1. Each production rule creates an (abstract) class.
2. For each alternative, a subclass of the production rule's class is created.
3. Every non-terminal is mapped to a containment reference of the created (abstract) type of the production rule(s) for that non-terminal.
4. Every terminal is mapped to an attribute.

The mapping is straightforward, just the alternative to subclass mapping might surprise at first glance. Choices and subclasses both represent exchangeability. The 3. and 4. steps are mixed to contain the symbol order.

Mapping Metamodels to context free grammars adds the problem of representing cross-references for which there is no correspondence in BNF. They can however be represented as terminals identifying the referenced element.

This mapping defines the foundation to bridge between Metamodels and context free grammars. It describes how a model represents a Parse Tree. The Parse Tree is bound by the concrete textual syntax. In general, however, Ecore models describe the abstract syntax of a language and not the Parse Tree of one textual representation. To gain more flexibility, the EBNF description is enriched by an synthesized attributed grammar which defines the AST. The mapping between S-attributed grammar and Ecore will be described after a discussion of XTexts grammar.

#### Xtext Grammar

XTexts grammar language is a domain-specific language to describe textual languages. The main idea is to describe the concrete syntax and its mapping to an in-memory representation - the "semantic model". Xtext supports grammar mixins, which is the reuse of existing grammars. It can infer Ecore models from a grammar, but it's also possible to import existing Ecore models instead. In the following, the XText grammar will be explained by example in conjunction with the optional Ecore inference. The Metamodel created by inference aids in the understanding of how Metamodel and synthesized attributed EBNF constrain each other. In case of the manually maintained Metamodel, this Metamodel must follow certain constraints. Based on the Liskovs substitution principle, the classes of the manual Metamodel must be a substitute of the classes of the inferred Metamodel which would have to be created. In case of model inference, the created structural features are normalized, meaning if all subclasses share a feature with the same name, type and multiplicity, they are merged together in the common superclass.

XText uses an EBNF with the following notation extensions:

- Option, meaning one or none is denoted by `?`
- Any, meaning zero or more is denoted by the Kleene star `*`
- One or more is denoted by the Kleene cross `+`
- Keywords are enclosed in `'` or `"`, e.g., `"keyword"` or `'anotherKeyword'`
- Grouping is possible by enclosing brackets `(` and `)`, e.g. `( A b C)`

**Lexer Rules** In XText there are Lexer rules, which extend the EBNF notation by:

- *Character Ranges*, which are denoted by start character `..` stop character, e.g., `'0'..'9'`
- *Wildcards*, which allow any character are denoted by '`.`'
- *Negation*, denoted by `!`
- the *end of file* token, denoted by `EOF`
- *Until token*, which consumes everything until the stop token occurs is denoted by `->`, e.g. `'/*' -> '*/'`

A terminal rule looks like for example:

```
terminal SL_COMMENT : "//" !'\n'* '\n';
```

this is equivalent to

```
terminal SL_COMMENT returns ecore::EString : "//" !'\n'* '\n';
```

The return type is optional and by default `EString`. The order of terminal rules is important, because they shadow each other. Terminal rule names are written in capital letters as a naming convention. Terminal rules are mapped to the data type following the returns keyword and are converted to it using the `ValueConverters`.

**Parser Rules** A parser rule produces a tree of terminal and non-terminal tokens and all parser rules together produce the Parse Tree, which is also referred to as Node Model in Xtext. Additionally, parser rules are the building plan for the creation of `EDataTypes` or `EObjects` which form the AST. In XText the AST is also called semantic model. `EObjects` are created lazy by default. Actions and assignments are used to derive types, to force their instance creation and to build the AST accordingly. The first parser rule in the grammar is the start production. It's possible to hide terminal tokens from the parser, like comments. Hidden tokens are ignored in the parser, but should be preserved during deserialization.

**DataType Rules** `DataType` rules are parsing-phase rules, which return `EDataTypes`. They are similar to terminal rules, but because they are parser rules, they are context sensitive. Rules that don't call parser rules, nor contain actions or assignments are considered data type rules with the default return type `EString`.

```
Number returns ecore::EInt : NUM ('.' NUM*)?;}
```

**Assignments** Assignments are the synthesized attribution of the EBNF dialect. They assign the information of a consumed symbol to the structural feature of the currently produced object, e.g.:

```
Person : "P" age=Number;
```

This creates a Metaclass named `Person` with an attribute named `age` of type `EInt`. The type of the current object(a subtype of `EClass`) is the return type of the parser rule. The type's name is equal to the rule's name by default.

```
Person : "Person" ("alias:" aliases+=ID)+;
```

This rule creates a multi-valued structural feature for `aliases` and adds each ID occurrence to it. Also `?=` exists which assigns true to a boolean value if the right side was consumed.

```
Class : "class" (public?="public")?
```

The right hand side of an assignment can be a rule call, a keyword, a cross-reference or an alternative composed by the former. It is also possible to specify the type of the returned `EObject` of a rule explicitly using `returns`. The previous rule is equal to:

```
Class returns Class : "class" (public?="public")?
```

**Unassigned Rule Calls** If a rule does not contain any assignments, but calls Parser Rules, it's called an unassigned rule. The return type of this rule is an abstract class, which has to be a supertype of returned types of `TokenA, TokenB` and `TokenC`. Xtext generates an abstract class named `AbstractToken` for this example by default.

```
AbstractToken : TokenA |        TokenB |        TokenC
```

**Linking** Xtext allows the declaration of cross-references in the grammar as terminal tokens, but not the linking semantics because of its synthesized attributed context free nature. The actual resolving process is done programmatically and is not of interest for this thesis. The general idea is discussed in Xtexts Architecture in section 2.3.3.

```
Reference:  "Ref" event = [MyEvent| VALID_EVENT_ID]
```

In this example, it's important to note that `MyEvent` is the referenced types `EClass`, not a parser rule. The `|VALID_EVENT_ID` is optional and references to an `EDataType` rule or uses a string by default.

**Enumeration Rules** Enumeration Rules convert between enumeration literals and strings and are a shortcut for data type rules with specific value converts.

**Actions** The returned object with its type can be explicitly controlled with actions. Xtext supports two kinds of actions: simple actions and assigned actions. Simple actions explicitly instantiate the `EObjects` type of the current choice using the notation `{EClass_aka_EObjectType_To_Return}`.

```
Z       :       {B} "A" v=ID
        |       {C} "B" v=ID;
```

Xtext requires and infers that `B` and `C` are subtypes of `Z`.

Assigned actions are used for tree rewriting which is necessary to cope with shortcomings of the underlying parser technique. Because XText uses the ANTLR parser generator framework, which is based on the LL(*) algorithm, it can't handle left recursive grammars. It is therefore necessary to rewrite or 'massage' the grammar, which is called "left-factoring" in the case left recursion is dissolved. Because this left-factoring ends in an unwanted AST, it is possible to rewrite the tree by assigning the `current EObject` to a structural feature of the returned `EObject`, e.g.,

```
Expression      :       LExpression
                ( "&&" {And.left=current}  right=Expression)
```

whereas `And.left=current` is the tree rewrite action which assigns the element currently-to-be-returned (`current`) to the structural feature `left` of the newly created element `And`.

**Syntactic Predicates** XText offers the possibility of syntactic predicates, which are necessary to solve ambiguity, like the dangling else problem and also help in cases where grammar rewriting would be necessary otherwise. Syntactic predicates are a specialization of semantic predicates regarding the underlying parser generator framework ANTLR. Syntactic predicates enable backtracking for a local scope and are enabled in XText by `=>`.

### Xtext Architecture

This section describes a subset of Xtexts architecture. It focuses on parsing, Parse Tree construction, the gap between abstract syntax and Parse Tree and linking.

XText converts the input grammar to an EMF model, saves it and provides helper methods for runtime grammar access. XText integrates itself in EMF as a `Resource` named `XTextResource`, which is shown in figure 2.3. The basic components of a XText resources are the `Serializer`, to convert `EObjects` to text, the `Parser`, to convert text to `EObjects` or to deserialize the `EObjects` and the `Linker` which manages cross-references.

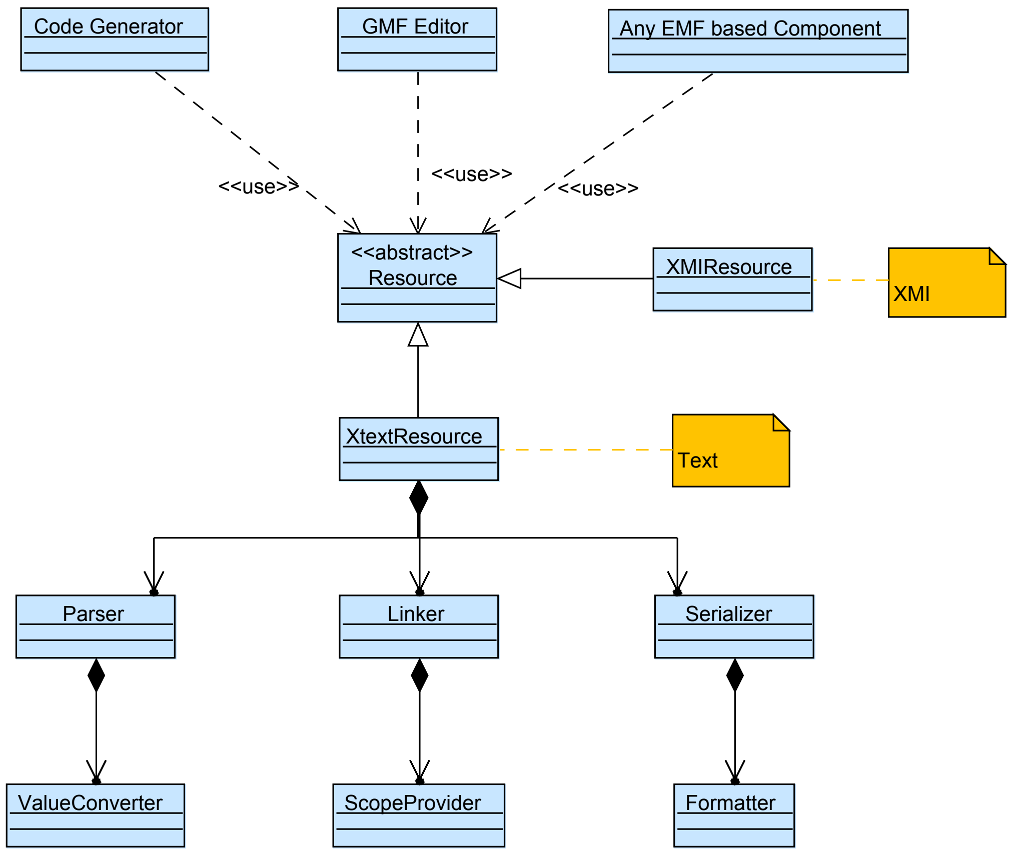

Figure 2.3: XtextResource Implementation, based on [XTe12a]

The main drawbacks of this solution are:

- During incremental changes, the XText parser replaces subtrees instead of updating them, so existing `EObjects` become stale. The XText documentation advises against the use of a “self-synchronizing (...) editor” on the same model as XText but suggests working on a copy.
- The implementation to identify `EObject` in a `Resource` has to return stable fragments.

**XText Phases during deserialization** The parsing process in XText is separated into the following phases:

1. Lexing and Parsing: The lexing phase transforms a sequence of characters into tokens. Tokens are a strongly typed part of the input sequence. They represent an atomic symbol and their type is determined by a particular terminal rule or keyword. The parsing phase requests tokens and builds a Parse Tree and an AST according to the grammar rules.
2. Linking resolves the textual identification of referenced targets to cross references after parsing is complete.
3. The validation phase is optional.

**Value Converters** Because Xtext translates between an abstract syntax tree and a stream of characters in both directions, the sequence of characters inevitably has to be converted to a proper data type and vice versa. This bidirectional transformation is the job of `ValueConverters` and is done by two (inverse) unidirectional transformations. Converters are used by the lexers terminal rules and parser rules.

**Parsing** XText uses ANTLR [Par07] for parser creation. It creates ANTLR grammar files with ANTLR grammar Actions to create the AST and Parse Tree. The rules return `EObjects`, ’init’ and ’after’ actions are used as guidance of the current stack position for the creation of nodes. The returned `EObjects` form the AST and relate to them and a Node Model is created which is related to the AST.

**Node Model** The Node Model is created during parsing. The term “Node Model” is defined by Xtext and does not conform to the model definition of this thesis, because no Metamodel exists. The base interface of the Node Model is `INode` which is described as “A node in the Parse Tree” [XTe12b]. Nodes are either composite nodes if they represent non-terminals or leaf nodes if they represent terminals. Nodes hold a reference to the grammar element they represent and leafs to the represented token additionally. Nodes are added to the `EObject`s as adapters instead of referenced to with an `EReference`.

The roles of the Node Model during serializations are:

- preserves existing white spaces
- preserves existing comments
- preserves the representation of cross-references, in case multiple names are possible
- preserves the representation of values, in case multiple representations are possible, e.g., 1.0, 1.00, 1.000.

**Linking** Linking of cross-references by means of textual identifiers can be separated into two parts:

- how XText copes with the mismatch between cross-references and text in general. In this case, just inner resource references are considered.
- how does `XTextResource` cope with integration in EMF as an EMF resource.

The general idea of linking in XText is that every grammar rule which contains a reference is visited after the ANTLR parser finished. While visiting the AST, an `ILinker` instance is consulted to resolve each link. How the link is resolved is specified by the language designer by programmatically using the textual representation saved in the Node Model and the context, which is the `EObject` holding the reference which has to be resolved. XText strongly encourages the usage of lazy linkers, which replaces the link with an EMF proxy containing the proper URI.

Besides the option to export just a subset of XText’s created `EObjects`, to achieve full EMF integration every `EObject` has to be referenceable, so the resource must be able to return the URI fragment of the given contained `EObject` and the `EObject` for a URI fragment. By default, XText’s Fragment Provider uses path fragments like

`//@classes.0/@methods.0/@localVariables.3/@name`. XText's documentation states that path references are fragile and that UUIDs should not be used regarding the users.

**Concrete Syntax Validation Constraints** As mentioned in [APCS03], it is possible to create invalid Metamodels, where no valid textual representation can be deduced, even if the Metamodel was inferred from the EBNF. The concrete syntax validation constraints describe how the grammar rules constrain a model, so it can be turned into word. They are used to validate if a textual representation is possible and also used to guide the serialization process to distinguish between different production and grammar rules, e.g.,

```
Z        :   "A" v=ID
         |   "B" n=INT;
```

given an instance of `Z`, it depends on if `v` or `n` is assigned to properly serialize `Z`. Given the rule

```
MyRule: ({MySubRule} "sub")? (strVal+=ID intVal+=INT)*;
```

several constraints are implied:

1. *Types*: just instances of `MyRule` and `MySubRule` are allowed. Further specialized Subtypes are prohibited.
2. *Features*: if other structural features beside `strVal` and `intVal` exist, they must be either transient, unassigned or contain the default value.
3. *Quantities*: the size of `strVal` and `intVal` must be equal.
4. *Values*: all values must be convertable to valid terminal tokens.

These constraints are created per production rule, not only per grammar rule. In the case of `Z`, two independent sets of constraints are created, one for the production starting with `"A"` and one for the production starting with `"B"`. XText's concrete syntax validation is not capable of regarding:

1. assigned actions, e.g., `{MyType.myFeature=current}`
2. unassigned rule calls to assigned actions
3. the order within list features, e.g., In `Rule: (foo+=R1 foo+=R2)*` foo must contain `R1` and `R2` in alternating order

**Serialization** The serialization process complements parsing and lexing by transforming EMF models into its textual representation. The XText documentation mentions six relevant steps:

1. Validation
2. Matching model elements with grammar rules by the Parse Tree Constructor
3. Associating comments
4. Associating existing nodes
5. Merging white spaces and line wraps
6. Adding white spaces by the formatter

In this thesis, only the Parse Tree Constructor is discussed because:

- Validation is optional and is done with less meaningful error messages by the Parse Tree Constructor.
- Comments, white spaces, line wraps are layout information without grammar relation.
- Runtime analysis showed that the existing Node Model is already accessed by the Parse Tree constructor, which indicates a gap in documentation and implementation.

**Serialization contract** “The contract of serialization says that a model which is saved (serialized) to its textual representation and then loaded (parsed) again yields a new model that is equal to the original model.”[XTe12a]

The following must hold:
I) $load(save(model)) \rightarrow model$
The inverse does not always hold, thus:
II) $save(load(text)) \nrightarrow text$
If the Node Model still exists, II) holds in most cases.

Given the XText example, that for the following rule first all `xvals` and then all `yvals` will be written to the token stream.

```
MyRule: (xval+=ID | yval+=INT)*;
```

This indicates that XText follows the `EObject` containment order instead of the one of the Node Model.

**Parse Tree Constructor** In actuality the Parse Tree Constructor does *not* construct nodes of the Node Model, it finds valid grammar rules for each `EObject` of the AST.

It succeeds if all `EObjects` are visited and no concrete syntax validation constraint is violated. Therefore, the Parse Tree Constructor has to find a valid allocation for the traversed AST in which all grammar constraints hold. This is, like parsing, a constraint satisfaction problem. XText solves this constraint satisfaction problem using top down backtracking, which results in potential exponential runtime of the serialization of a model. This is done by the Parse Tree Constructor finding valid 'paths' from the root to a leaf node constrained by the grammar. The Parse Tree Constructor stops after the first valid solution has been found.

If there is no representation of a choice in the AST, a default needs to be specified, for example, in the case:

```
PluralRule: "contents:" count=INT Plural;
terminal Plural: "item" | "items";
```

it is unclear during Parse Tree construction if 'item' or 'items' should be used. This case is named unassigned text in XText.

### TEF - Textual Editing Framework

With TEF it is possible to integrate textual editing in an existing MVC editor. This is done by selecting an element, showing its textual representation in an extra overlay and executing the changes when the overlay is closed. The basic steps are:

1. Copying the selected element and all its directly and indirectly contained elements.
2. Creating an initial representation in an overlay text editor using backtracking.
3. Using background parsing to constantly create new models at the location of the copy.
4. Replacing the edited model element if the overlay was closed.

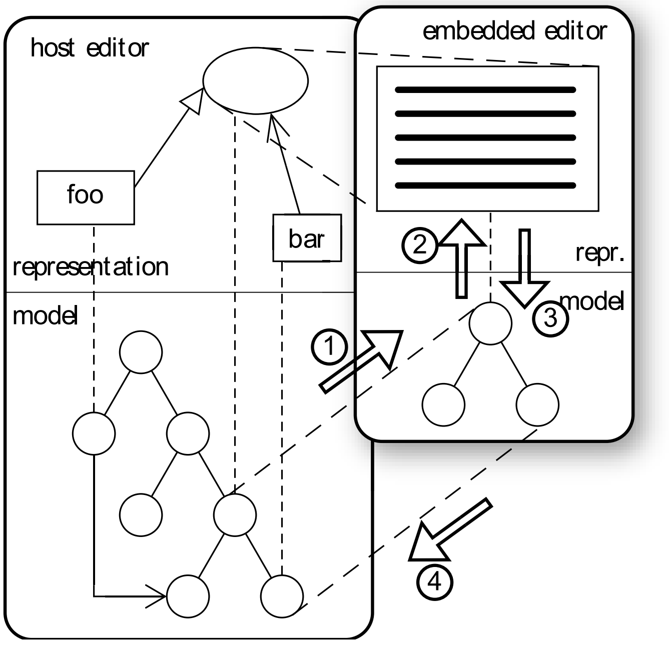

Figure 2.4: Steps involved in the embedded textual editing process, from [Sch08]

Replacing has two fundamental problems:

- All references to the replaced element break.
- Information which is contained in the original model, but not displayed in the textual representation, is lost.

In order to solve the problems by reassigning the references and merging the models, identification is required. [Sch08] demands language-specific identification.

TEF does not require the whole model to be describable textually, only the elements and it's directly or indirectly contained elements which should be textually displayable must be specified using a context free grammar.

TEF creates a Parse-tree from the model by traversing the model along its composition and determines a set of suitable grammar rules based on constraints. It uses backtracking for that and [Sch08] also suggests to enable the language engineer to prioritize grammar rules if multiple Parse Trees are possible.

# 3 Methodology

This section describes how the results were achieved. The overview figure 3.1 presents the proceeding steps.

The following observations during research were made:

- Every non Metamodel based tool uses its own, specific approach for abstract language data structure transformation, validation, persistence and editing.
- In the Graphical Modeling Framework (GMF), the *Notation model* generically encapsulates the state of the visual representation.
- After a short evaluation of Meta Programming System (MPS), the restricted usability of *projectional textual editing is undesirable* for the target of this thesis. The text editing concept of MPS is template or form based, not free textual. This concluded in additional research.
- The CUP2 [1] Parser generator framework was evaluated to enable the direct use of `EObjects` in the Parser. The insight, that Structured Tokens and thus `EObjects` as tokens are valid, led to the abandoning of CUP2.

[1] http://www2.in.tum.de/~petter/cup2/

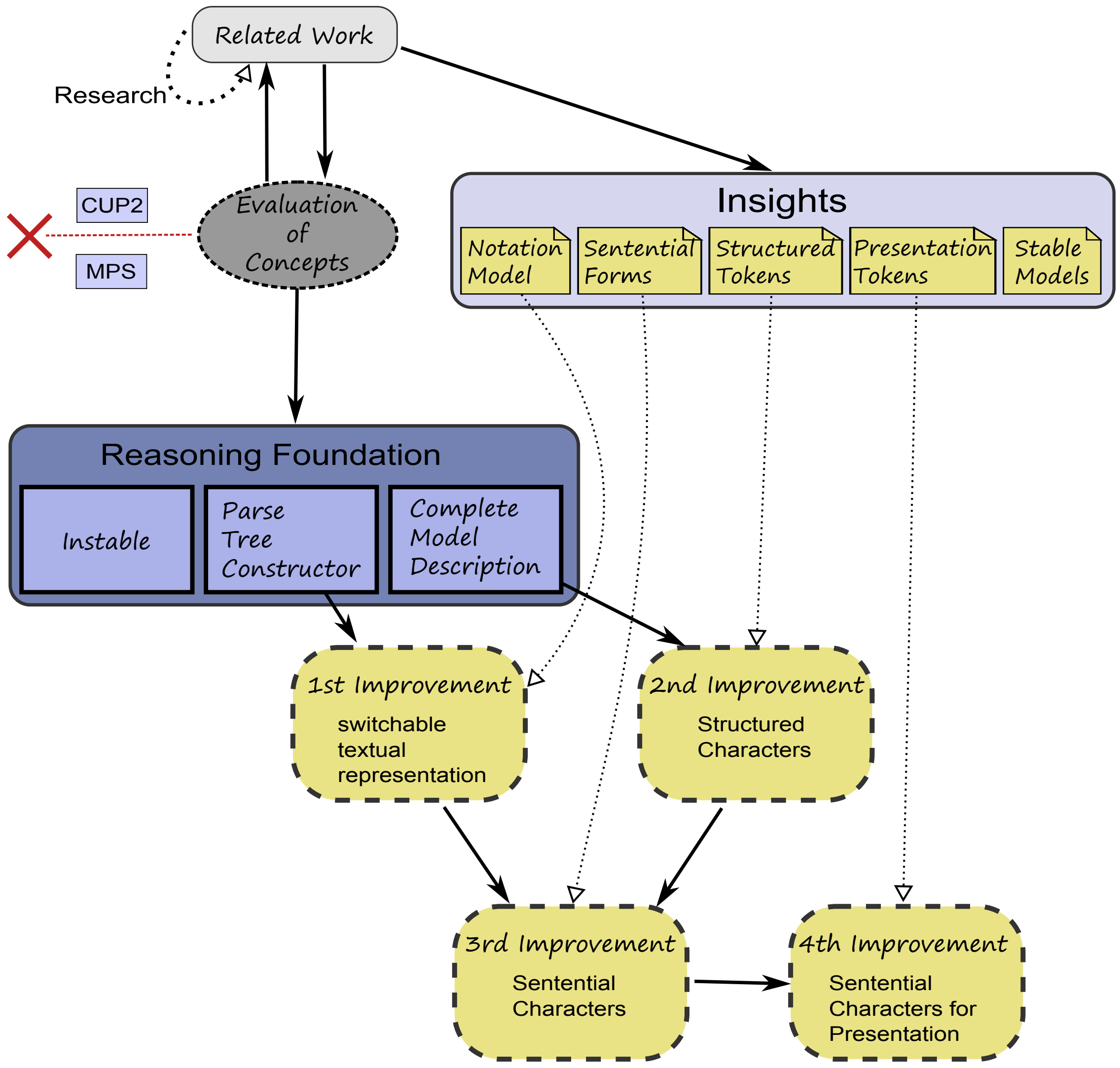

Figure 3.1: Methodology Overview

- Proxima uses a *Structured Token for graphical presentation*, but does not allow it to be directly edited. Its seven layer architecture with bidirectional model transformations between each allows high flexibility. Dependant layers were considered undesirable, especially regarding possible user interaction on different layers, with the complexity to maintain bidirectionality by two unidirectional transformations and updating instead of replacing model elements. Several model transformation languages for EMF exist, so structural transformations are possible and separated if required.
- Harmonias incremental parsing algorithm for GLR parsing requires the parser to handle *sentential forms*.
- The report [KM06] about Barista shows the successful use of a variant of Harmonias incremental parsing algorithms in projectional editors to achieve *stable abstract data structures*.

The Xtext framework was evaluated afterwards in respect of adding alternative representations. The following central perceptions were made:

- The used, non incremental parsing technology results in unstable models. This characteristic is not considered until its impact on the solution in chapter 5 discussion.
- The grammar must completely describe the model.
- There is no isomorphism from grammar production to `EObject`, which means multiple representations of the same `EObject` are possible.
- The Parse Tree Constructor determines the production used for model to text transformation. Different textual representations are already possible, but are not accessible by the user or language designer.

Due to problems with model stability and the gained insights of extensive research, the argumentative deductive approach was chosen for this thesis. The perceptions made for the Xtext framework are used as the reasoning foundation.

The first conceptional improvement of this thesis allows *switchable textual representations*. This has been solved by extending the Parse Tree Constructor and introducing a Notation Model.

*Structured Tokens* highlighted the usage of nested structured data as tokens, which triggered the second improvement. To integrate Structured Tokens in a textual serialization, a Structured Character concept was developed that enables the use of `EObject`s as atomic characters on the character stream.

The Notation Model developed for the first improvement and the possibility of Structured Characters by the second, together with grammar post-processing enables Sentential Characters.

The forth improvement enhances Sentential Characters for presentation purposes.

# 4 Results

This chapter begins with an analysis of Xtext to identify possible starting points for alternative representations and obstacles. Then an outline of the solution is presented followed by a minor extension of Xtexts Parse Tree Constructor, the Guided Parse Tree Constructor. In the next two sections, Metamodel for an attributed grammar and a depending Notation Metamodel are developed. The Structured Characters section outlines a solution to facilitate structured information at position of a single character. The last section focuses on their combined use to enable alternative graphical representations.

## 4.1 Xtext Analysis

This section analyses Xtext beyond the scope of Xtext's manual [XTe12a]. At the beginning, isomorphism relations of Metamodel and grammar is discussed. Insights about isomorphism explain the variety of valid Parse Trees for a model. This leads to further insights in the Parse Tree Constructor. Subsequent peculiarities of Xtext's grammar are mentioned, followed by deficits of XtextResource and Xtext's Node Model for the desired solution.

### 4.1.1 Model Grammar Isomorphism

The explicit return types decouple the `EObject` type from the grammar rules in a well defined manner. Simple actions already made this possible, but the subtype of return type constraints strongly limits its useability. Return types raise the abstraction level, because an `EObject` of the abstract syntax can have a context dependent representation. It also allows multiple representations of the same `EObject` type, so this has a special significance for the Parse Tree Constructor.

On the other hand, this changes the time complexity of Parse Tree construction from $O(N)$ to $O(c^N)$ of the current XText implementation. The following explanation of choice mappings hint the problem, but it is explained in further detail in the Parse Tree Constructor at 4.1.2:

```
Z        :   "A" v=ID;
```

this rule creates an `EClass` `Z` with an `EString` attribute `v`. Isomorphism is kept between the type `Z` and the production rule.

```
Z        :   "A" v=ID
         |   "B" n=INT;
```

creates an `EClass` `Z` with the attributes `v` of type `EString` and `n` of type `EInt`. Isomorphism of this type to production rules is lost, but might be regained by a constraint if `v` or `n` is assigned. Isomorphism to a grammar rule is kept.

```
Z returns A : "A" v=ID;
Y returns A : "B" v=ID;
```

No isomorphism of the type `A` to a grammar rule, because both rules return an `EObject` of type `A`.

### 4.1.2 Parse Tree Constructor

This Xtext Parse Tree Constructor section continues the Parse Tree Constructor paragraph 2.3.3 of related work by two descriptive examples.

**First Example** For example, if the following grammar parses `somekeyword 0 C`, the AST in figure 4.1 will be constructed.

```
S        :        v=A
         |        v=X;

A returns Obj   :        l=B r=C    ;
X returns Obj   :        l=Y r=Z    ;
B returns N     :        "somekeyword"    v="0";
Y returns N     :        "otherkeyword"   v="0";
C               :         {C} "C" ;
Z               :         {Z} "Z" ;
```

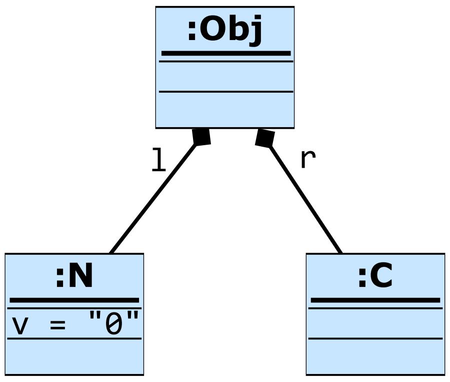

Figure 4.1: AST for "somekeyword 0 C"

To create this Parse Tree without a Node Model, the decision if the rule of the root `Obj` is `A` or `X` can just be decided after the type of it's right child `C` is determined. It is possible that all grammar rules return the same type, so type information would be useless to guide production rule resolution.

**Second Example** In the second example the nondeterministic finite automaton for the following rule is presented in figure 4.2. The Parse Tree Constructor determines all non default values of `EObject` of `AClass`. It then tries to find a valid path from the top to the `stop` node. A path is valid if a matching non default value is consumed at a constraining node and if no non default value is left when the `stop` node is reached. It is obvious that if `j` would have left out, two paths would have equal constraining nodes. In a valid scenario this consuming process is subsequently done for every model element from the models root downwards until a tree of valid paths is found which successfully visited all `EObjects`. Xtext skips Parse Tree construction and directly prints all nodes, including production nodes, on each path to the character stream. This assumes a certain associativity.

```
A returns AClass:        "option1" v=INT "post"
                                |        "option2" v=INT j=ID
                                |        "ID"      i+=ID+;
```

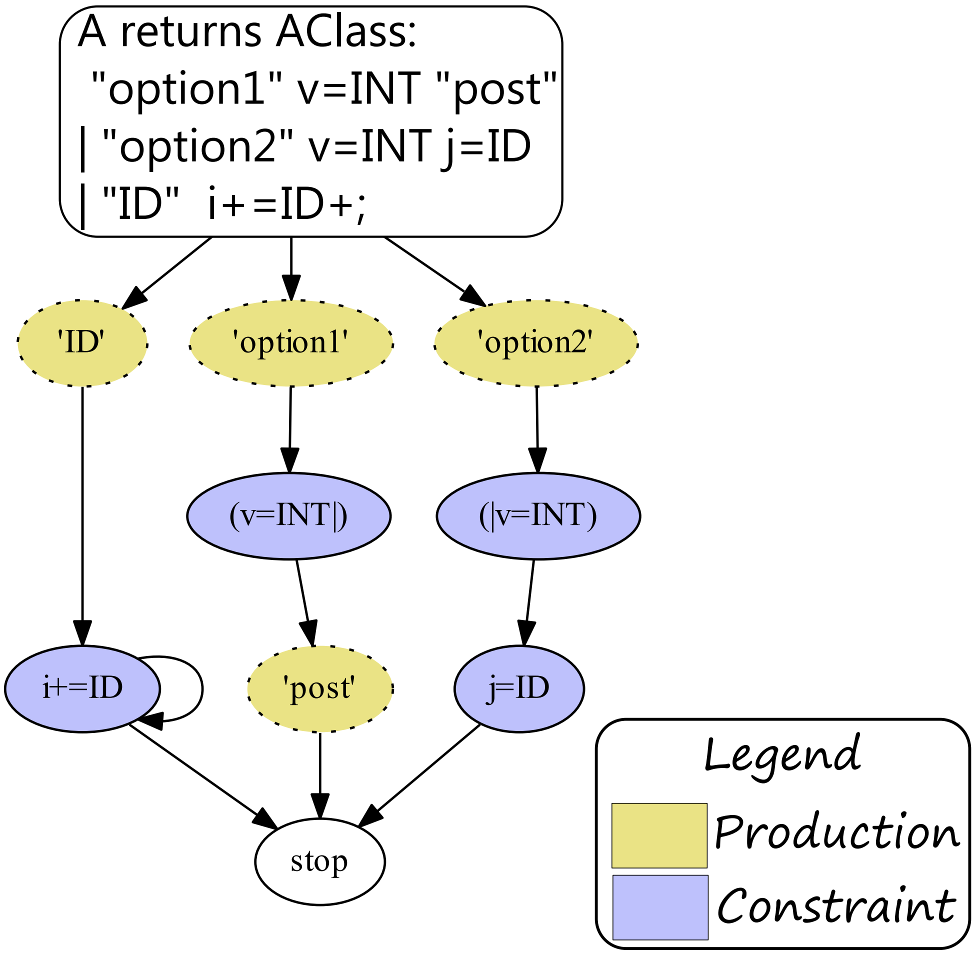

Figure 4.2: Second Parse Tree Constructor Example

### 4.1.3 Grammar

In the XText grammar, it is not possible to statically assign a value to a structural feature. For example, to assign `0` to an integer attribute `v` of class `MyBoolean` in the `"false"` case and `1` in the `"true"` case requires writing a special `ValueConverter`.

```
MyBoolean:  "true" | "false"
```

A much greater impact on the grammar is it's inability to express left recursive grammars and the consequences. This is owed to the LL(*) parsing algorithm of ANTLR, the parser generator used by XText. This requires left factoring and grammar rewriting which lead to XText allowing tree rewriting of the AST by assigned actions. If only parser and AST construction is regarded, this is more or less an inconvenience for the grammar designer who needs to left-factorize, but it creates problems regarding Parse Tree construction and validation for XText's current implementation. Considering that the current Parse Tree construction algorithm has a runtime of $O(c^N)$, it is arguable to use a GLR parsing algorithm with $O(n^3)$ to avoid the need for grammar massaging and thus AST rewriting.

### 4.1.4 XtextResource

XText is, like EMFText [EMF12], integrated in EMF as an `Resource`. The responsibility of resources is to serialize and deserialize models, which leads to various problems:

- The grammar must describe the whole model: the model must inform what is not regarded by the grammar, except for `volatile` information. Adding additional information, like an ID, is impossible without incorporating it in the grammar. To circumvent that restriction, the whole model must be textually described. A possible solution would be to create additional `EObjects` in an additional `Resource`, which points at the `EObject` in the `XTextResource` which should be enriched. With an `ECrossReferenceAdapter` the inverse references and thus the additional information of the extended object can be obtained. This technique to extend an `EObject` in a non-invasive manner is used to implement UML2 stereotypes in eclipse UML. This concept depends on references and their integrity.
- The XText editor edits the text file, meaning it edits the serialized form of the model, not the actual model in the memory. In conjunction with a non-incremental parser, model elements are replaced instead of updated. Furthermore, EMF's `ResourceSet`, which keeps referential integrity, is bypassed. XText demands the programmer to handle referential integrity or to "return

stable fragments for its contained elements" [XTe12a]. For example, the expression `int i=0` in a programming language does not contain an intrinsic identifier, so this is impossible for an arbitrary language. UML2 solves the problem of referential integrity by assigning every model element a universal unique identifier (UUID). These UUIDs are handled by the resource and are externally attached to the model objects. To keep referential integrity, either modifications must be done in a ResourceSet or extrinsic UUIDs must be used.

- Because the model contained in a `Resource` is not modified in memory and extrinsic, non-grammar conform information like IDs can not be added or integrated persistently to model elements, referential integrity can not be maintained. On the other hand, the determination of changes needed to update a model without unique IDs is based on heuristics, thus potentially inaccurate. To enable proper updates and keep referential integrity, editing must not be done on the textual serialized form of a model. This does not contradict the ability to store the model in a textual form for, e.g., viewing and versioning.

### 4.1.5 Node Model

The potential use of the Node Model is strongly restricted in Xtext, for the following reasons:

- The Node Model is not an EMF model.
- The Node Model is not explicitly available without running the XText parser, because it is created by the parser from information of the concrete serialized language model.
- The use of the runtime instances of the Node Model is restricted by the API: "clients should never keep a reference to a node as it may be invalidated at any time and the very same object could be reused in another subtree of the full parse tree."[XTe12b]
- The Node Model is not updated during Parse Tree construction. If an update is required, an additional parse with its problematic replacing instead of updating behavior is necessary.

- Even if the Parse Tree Constructor would construct the Node Model, it takes the first valid solution. It is not possible to choose between different valid representations or prefer a valid one which are semantically equivalent, e.g.,

```
Foreach                  :       Map | For;
Map returns FE           :       "map"           v=ID;
For returns FE           :       "foreach"       v=ID;
```

## 4.2 Outline of the Solution

The general idea of the presented solution combines three simple concepts. The first two already allow a potential editor to operate on a Parse Tree part:

- The use of private characters as a unique key for a mapping, which contains `EObjects`. The value of the map is an object, thus a single character can be resolved to an arbitrary data structure.
- The Parse Tree is a tree data structure. If a part of the tree produced by a token sequence is saved in a single token, that token is a valid substitute for that part of the tree.
- A Parse Tree constructor, which is extensible in regard to assign types to and to skip Parse Tree construction for invisible elements. This allows the graphical editor to use the abstract syntax tree directly as a data source.

Figure 4.3 shows the conceptional overview. On the right side of the figure are the classical parser related elements, from below upwards:

- a character stream
- a lexical analyzer or lexer, which reads the characters and groups them into tokens.
- the token stream, which is produced by a Lexer and used by a Parser to create the "Parse Tree"

On the left side is a language model, which is deduced from the Parse Tree by the attribution of the grammar. In parser literature, *this language model is called an Abstract Syntax Tree.* The use of a modeling term instead of parsing term emphasizes where models are integrated and that language descriptions in modeling concepts

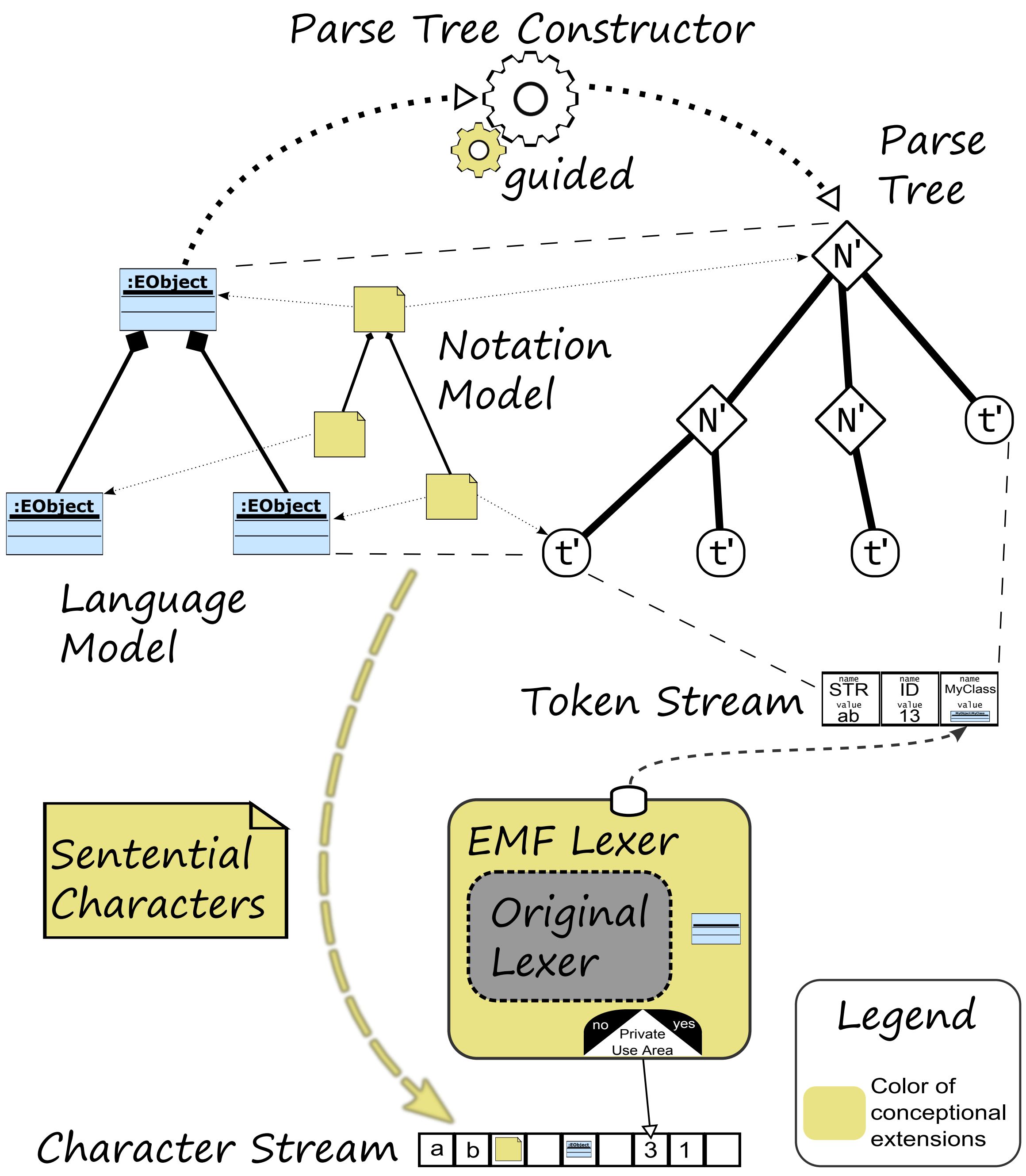

Figure 4.3: Conceptual Overview

are abstract. In contrast to context free grammars, a language in modeling terms lacks a concrete notation.
The inverse transformation, from language model to Parse Tree is done by the Parse Tree Constructor. This transformation is ambiguous, because for one language model more than one valid Parse Tree can be constructed.
The described architecture is implemented in XText. This thesis conceptually extends the architecture by extending the Parse Tree Constructor, by the introduction of a *Notation Model* and an *EMF Lexer*.
The Parse Tree Constructor is extended by:

- Adding valid alternative constructions to the Notation Model and considering them while constructing the Parse Tree.
- Extensible rule evaluation capability, which adds a specialization of valid production paths to the alternative representations, depending on constraints.
- Omitting Parse Tree branch construction for flagged Notation Model elements.

The Notation Model is introduced to:

- be a serializable model.
- to add an approved design of graphical editors to separate language elements and their corresponding notation state.
- to guide the Parse Tree Constructor to be able to unambiguously construct a certain Parse Tree. This leads to a model which was in large parts similar to the Parse Tree itself, so the Notation Model was extended to also *substitute the Parse Tree* while still solving its guiding purpose. Figure 4.3 shows the Notation Model separated from the Parse Tree as a bridge between language model and Parse Tree because this integration feature is the advantage of the Notation Model over the Parse Tree. The degree of overlapping functionality of the Notation Model and the Parse Tree did not justify separated data structures.

The EMF Lexer is introduced to:

- separate private use characters from the character stream used for the original lexer.
- resolve the data structure identified by a private use character.

- determine a token name by properties of the `EObjects` in that data structure. This assignment process can be customized by additional rules that traverse the resolved data structure.

## 4.3 Guided Parse Tree Constructor

The Guided Parse Tree Constructor is responsible to find all valid Parse Trees for the current language model and select the best tree, based on ranking criteria. The actual Parse Tree construction process is twofold:

- find all combinations of production rule applications that uniquely distinguish valid words and
- create a Parse Tree according to the best tree, with the most promising combination.

The suggested solution is close to the one implemented in Xtext. Xtext's solution, which is described in section 2.3.3, stops after finding one combination. To enable multiple representations, the alternatives need to be made available. This is done by the Notation Model, which is described in section 4.5. The Notation Model is able to be equivalent to a Parse Tree, but also allows the user to reference to following production parts, called hints. EMFs `ECrossReferenceAdapter` allow navigation of unidirectional references in both directions. This allows the use or reuse of Notation Model elements which reference to AST nodes just by getting all referencing `EObjects` and filtering a type.

**Hints** A hint is a reference to a following production. So with hints, a node in the Parse Tree is not only labeled by its non-terminal, but also by its production. Hints specify a single production without referencing to a Parse Tree part, which is important, if multiple productions are possible. Thus, hints do not referenceable to actual content but define the structure of the content. Regarding the Parse Tree Constructor, the AST is present, but it is ambiguous as to which structure to use to hold its contained data.

**Parse Tree Construction Steps** The basic steps of Parse Tree Construction are:

1. compute all possible solutions. This results in a forest of distinguishable Parse Trees.

2. determine the best tree. Which criteria compose the best tree are explained below.

3. produce a Parse Tree for the best tree. The best tree uniquely distinguishes a Parse Tree, but for example keywords terminal creation can be postponed to Parse Tree construction.

4. compress the forest. For each branch which branches the best tree, set the first node of the branch as an alternative representation of the best trees node it branches from.

**Ranking criteria** Criteria that determine the result tree are ranked from most to least important:

1. Use Notation Model hints, if they exist. Hints are likely set by the user and thus have top priority.

2. Number of similar productions to the previous parse or Parse Tree construction state, which is saved in the Notation Model.

3. User preferences to prefer certain alternatives.

4. Language designer preferences.

5. Number of default values used as negative criteria.

6. Number of direct `EObject`s required as negative criteria, thus prefer description of declaration.

The suggested behavior is, that the user does not specify the whole alternative representation, but selects an alternative. Let the Parse Tree Constructor create and display the new representation and iteratively refine the presentation.

## 4.4 Grammar

In this section, a Metamodel for a synthesized attributed grammar is developed to formally describe the required grammar relation of the Notation Metamodel. The described grammar is simpler than Xtext's grammar, especially in regard to Actions. First a Metamodel for a normal, non-attributed, grammar is defined, then an example instance of it is presented, finally the grammar Metamodel is completed by an attribution extension.

By convention, transient model elements are modeled as private static elements, thus with a leading `-` and are underlined. Metamodels are also models, so the short term model is used when the distinction is obvious from the context.

### 4.4.1 Grammar Metamodel

Figure 4.4 shows a Metamodel for an EBNF grammar. The Metamodel lacks the ability to define lexemes.
Compared to the definition of a context free grammar, the non-terminals are the set of `NonTerminalNames` of `Rule`, the terminals are the set of `names` of `TerminalType`, the start symbol is the `Rule` referenced by `Grammar` and the productions are implicitly described by the directly and indirectly contained elements of the `Rules`. The `NonTerminal` and `Terminal` elements in the grammar are just references to the real non-terminals and terminals. This definition overlap is owed to the fact that the same terminal may appear multiple times in productions, which must be distinguishable. The denomination distinguishes between declaring and referencing existence. For example, in the following rule, $b_1$ to be an instance of `Terminal`, `C` an instance of `NonTerminal` and $b_2$ another instance of `Terminal`, but referencing to the same `TerminalType`:
`A` : $\mathrm{b}_1$ C $\mathrm{b}_2$

The `Grammar` has at least one `Rule` and one `TerminalType`. The `Grammar` has exactly one `start rule`. The `Rules` have a name and contain exactly one element as their `rightHandSide`. This element might be either a `NonTerminal`, a `Terminal`, a `Sequence` or an `Alternative`. `Sequences` and `Alternatives` are containers for at least two `RHS-Elements`. `Sequences` are `a b c` or `(a b)+`, for example. `Terminal` and `NonTerminal` hold references to the unique type they represent. `Symbol` just

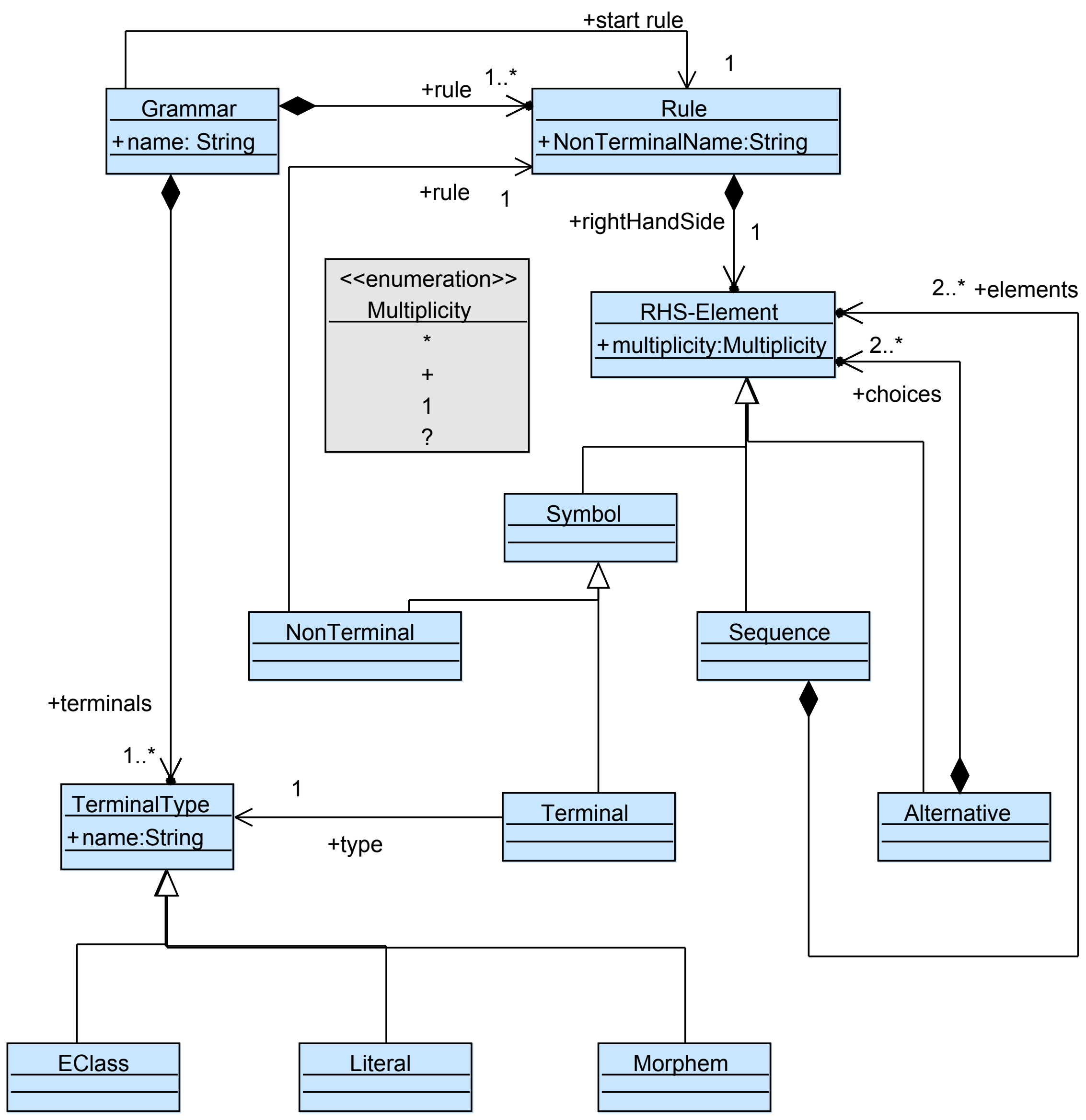

Figure 4.4: EBNF Grammar Metamodel

provides abstraction but does not add expressivity to the language itself. Every `RHS-Element` has a `Multiplicity`, so exactly one `1`, one or none `?`, zero or more `+` or any multiplicity `*` can be expressed. Subclasses of `TerminalType` are present to allow a more detailed specification of `TerminalType` in later models.

### 4.4.2 Grammar Example

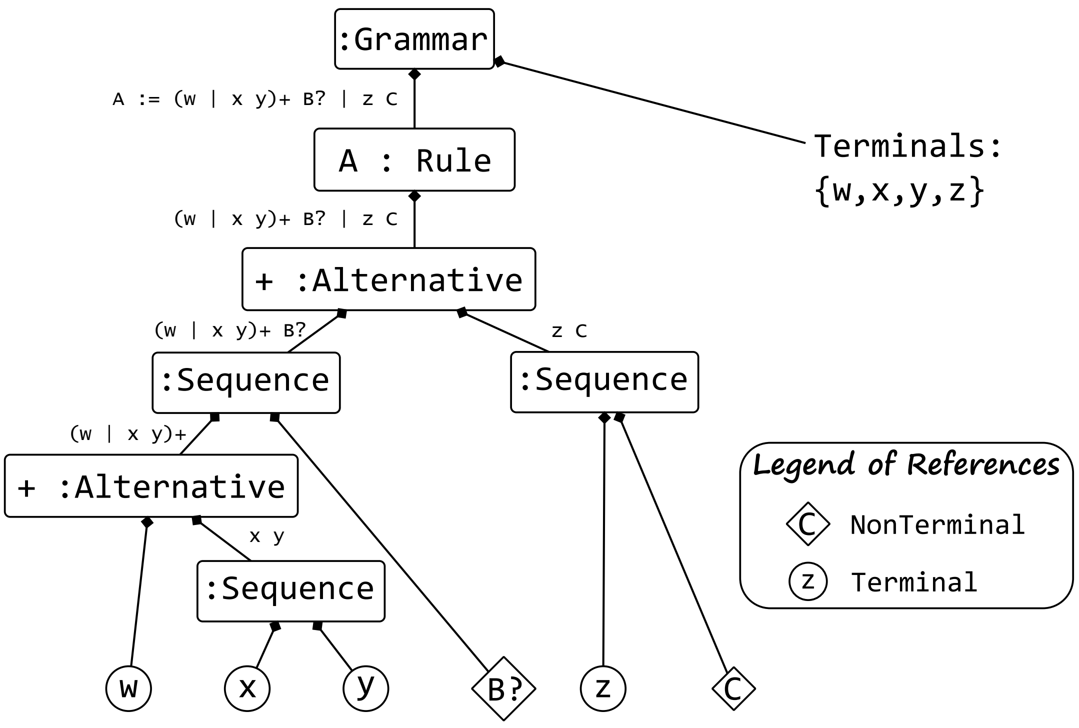

Figure 4.5: Example Rule "A := (w |x y)+ B? | z C"

Figure 4.5 shows the model instance of the EBNF Metamodel 4.4 of the grammar `A := (w |x y)+ B? | z C`. The grammar rules `B` and `C` are left out, as well as the start rule reference. The references to the symbols `w`, `x`, `y` and `z` are replaced with the name of the referenced symbol.

### 4.4.3 Attributed Grammar

The Metamodel defined in figure 4.6 adds attribution and default values to the grammar. It uses metaclasses from the Metamodel figure 4.4 for context free grammars. This separation is for documentation purposes only, because the metaclasses of the

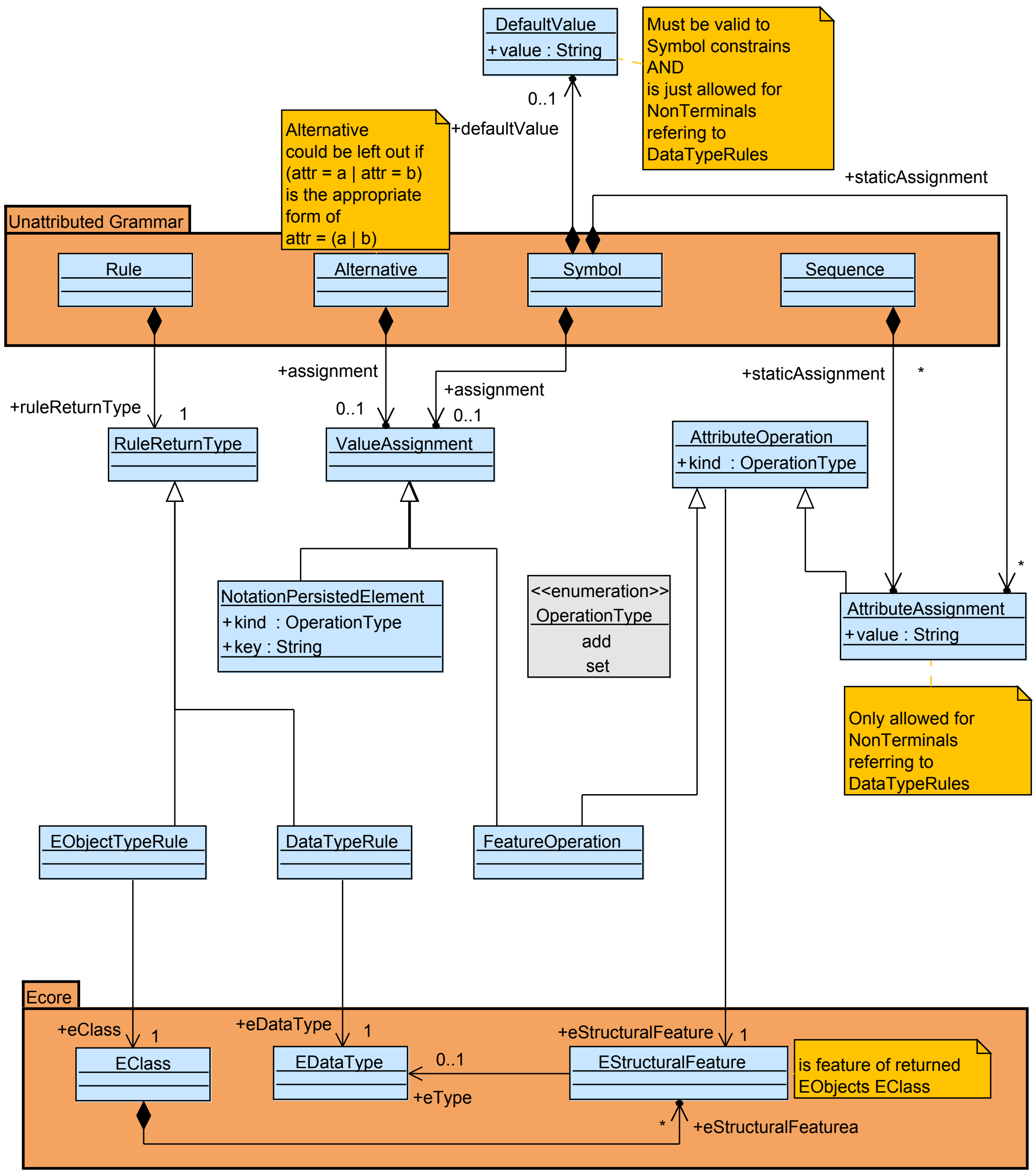

Figure 4.6: Attributed Grammar Metamodel extension

context free grammar Metamodel references to the current metaclasses.
Each `Rule` has a `RuleReturnType`, which might be an `EClass`, if it is an `EObject` returning rule or an `EDataType`, if it is a Data Type Rule. The additional distinction in `EObjectTypeRule` and `DataTypeRule` is for presentation purposes only, this makes it obvious in the diagram when an `EObjectTypeRule` is used. For a real implementation, a reference from `Rule` to an `EClassifier` would be sufficient. `EClassifier` is the supertype of `EClass` and `EDataType`. `Symbol` now can contain a `defaultValue`, which is a `String`. `Strings` are structureless, so the use of `DefaultValues` is restricted to `NonTerminals` referencing to `DataTypeRules` and `Terminals` only. The `String` must not violate the `Symbol`'s constraints. Given the example `attribute+=TerminalSymbol`, an attribute assignment is realized by `TerminalSymbol` containing a `FeatureOperation` with `kind` set to `add` and a reference to the `EStructuralFeature` named `attribute`. The `EStructuralFeature` must be contained in the `EClass` of the returned `EObjects` type. In contrast to Xtext, it is possible to assign statically a structureless value to an attribute, for example,

```
Rule : {ruleAttribute="true"} "1"
```

which means that if the rule matches `ruleAttribute` is assigned to `"true"`. This can be done for an arbitrary amount of `EStructuralFeatures`. `NotationPersistedElement` allows the persistance of a String in the Notation Model, this allows sharing of data between `Alternatives`.
`Alternatives` also contain a `ValueAssignment`. This could be omitted if

```
attribute= (a | B)
```

would not be allowed and instead only the alternative

```
(attribute = a | attribute = B)
```

would be allowed.

## 4.5 Notation Metamodel

In order to bridge the gap between a model and a token stream, a Notation Model is developed. The Notation Model is an EMF model, so it is possible to persist the Notation Model separately from the language model. The existence of a Notation Model is optional, after a token stream or a language model is created with one exception. The exception is that the use of morphemes is without default or language element representations. Due to this fact, they should be avoided. It is possible to create a Notation Model from a word of a language or from a model which complies to the additional constraints introduced by a grammar. The presented Notation Model does not include additional layout information, for example newlines, spaces, tabs or for visual editor's coordinates, styles, etc. In general, the Notation Model saves information which is not stored in the language model but relevant for presentation.

The main purpose of the Notation Model is to unambiguously describe a word which produces the language model and to enable a choice of different representations for grammar parts by providing hints for the Guided Parse Tree Constructor.

The main design considerations for the development of the notation Metamodel are:

- To be able to fully represent and substitute a Parse Tree, so token creation could be done easily.
- To support the creation of the Parse Tree from an AST. This means that it must be possible to guide the Parse Tree construction process by hints and hold alternative choices to choose. To be able to distinguish alternative productions that require granularity which is close to the one for Parse Trees.
- To have one notation element container per `EObject`. This does not only ease reasoning, but also allows stable paths if the corresponding notation element is a generic container. With the use of an `ECrossReferenceAdapter` for the referenced language model's resource, it is possible to obtain the corresponding notation element even if the structure of the Notation Model is lost.

Requirements:

- In contrast to Xtexts Node Model which is just updated during parsing and used to guide Parse Tree construction, the Notation Metamodel must be equivalent to a valid token stream and therefore describe it unambiguously.
- It must contain or reference to all information necessary to create a specific token stream together with a language model and a grammar model.
- It has to be a Parse Tree in order to create language elements if the corresponding tokens exist.
- Like in GMFs Notation Model, each language element should have a corresponding Notation Model element.

The diagram 4.7 shows the section of the notation Metamodel which is relevant for its function representing a Parse Tree and guiding Parse Tree construction. It is able to represent Parse Trees up to terminals, but does not contain the token values. The following two sections describe the intermediate steps which lead to this final model.

### 4.5.1 Evolving from a simple Parse Tree Metamodel

The Metamodel in diagram 4.8 is sufficient to describe a Parse Tree, where `TerminalNotations` represent the leafs and `NonTerminalNotation` the branches. It contains a subset of elements of the Notation Model, namely `NonTerminalNotation`, `TerminalNotation` and `ProductionPart`. The relationship of the subset elements differs from 4.7 in that `NonTerminalNotation` contains `ProductionParts` directly, but more importantly also that `NonTerminalNotation` can contain multiple `ProductionParts`. `NonTerminalNotation` and `TerminalNotation` are of a generic type, to determine which Symbol they represent, they reference to `NonTerminal` and `Terminal`. The distance[1] to the symbol is one, the distance to the symbol type is two. For example, in the following grammar, the type of the `b`s are equal, but the `b`s are not:
`S : b c | b d`
A verbose representation of this grammar would be:
`S :` $\texttt{b}_1$ `c |` $\texttt{b}_2$ `d`
The `TerminalNotations` reference to the exact position in the grammar, regarding the previous case for example to $\texttt{b}_1$, which refers itself to the type `b`. For an concrete parser implementation, this means that in case of a shift reduce parser, the reference

[1]between nodes in a graph

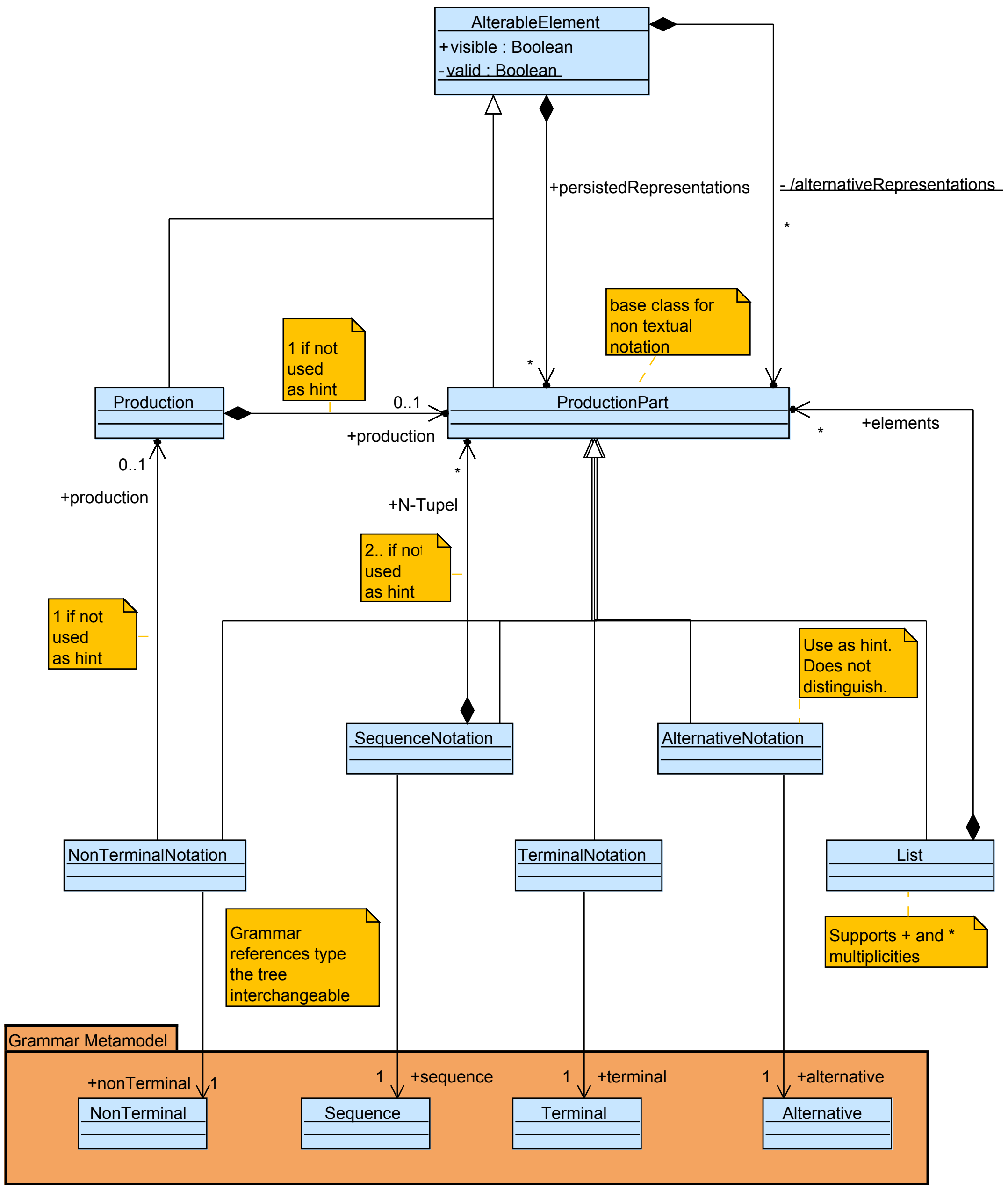

Figure 4.7: Production part of Notation Metamodels

to the NonTerminal can be at least one production later than normal, because the type is known when the rule is reduced, but the position is not known until its producing production is reduced. The exact positions are useful to guide Parse Tree construction and the referencing solution provides flexibility while keeping stable NonTerminalNotations and TerminalNotatations. The Parse Trees that are created by this Metamodel need post processing to group their structure.
For example, accessing the last b in the tree created by the grammar for the input b b b c b requires an iteration over each element:

```
A : b+ c b
```

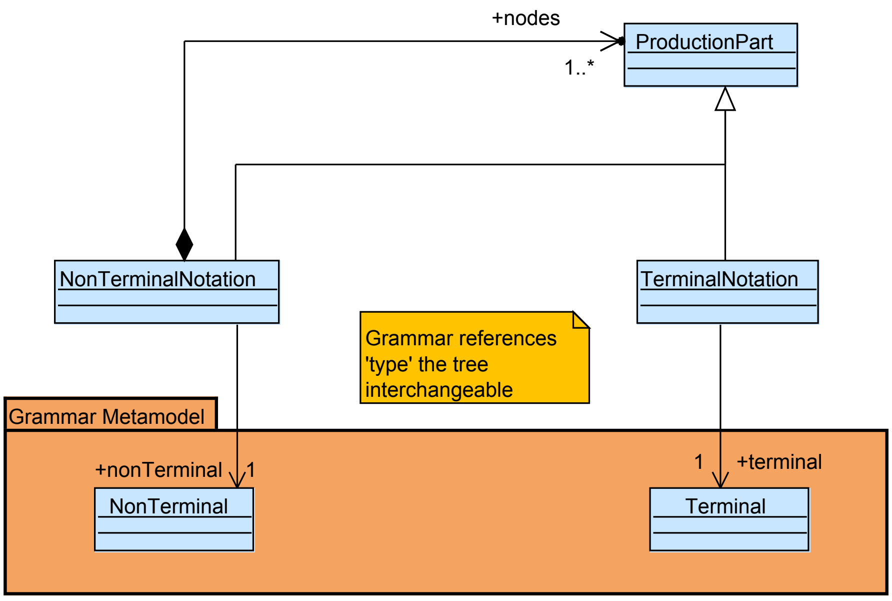

Figure 4.8: Simple Parse Tree Metamodel

## 4.5.2 Term Parse Tree Metamodel

A Metamodel able to hold terms structured is shown in 4.9. The NonTerminal-Notation holds exactly one reference to a ProductionPart named production. The distinction in different terms is held by SequenceNotation which contains an n-tuple of at least two ProductionParts. A List holds Symbols with a possible multiplicity higher than one. It might be beneficial to directly support multiple values per ProductionPart, for example a+, but this would not render List obsolete in order to support `(a | B)+`. Since alternatives do not occur in a Parse Tree, because one choice must have been used, they have no equivalent in the term Parse Tree Metamodel.

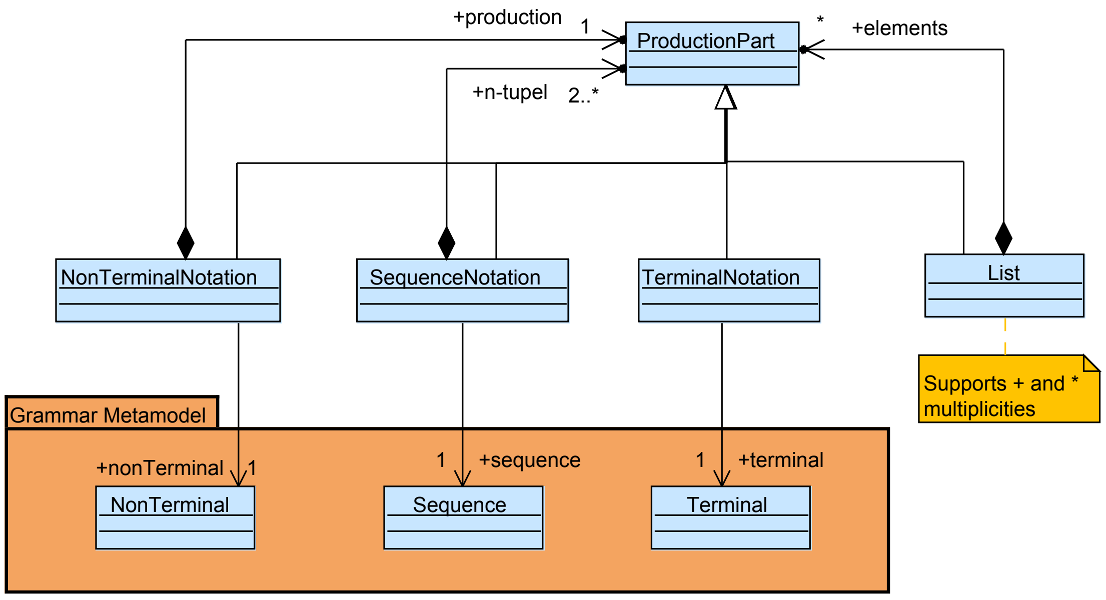

Figure 4.9: Term Parse Tree Metamodel

### 4.5.3 Production part of the Notation Model

Figure 4.7 shows the final production part of the notation Metamodel. It differs from the one used for terms in the following points:

- an additional indirection from `NonTerminalNotation` to `ProductionPart` was introduced.
- `AlternativeNotation` to represent `Alternatives` was added.
- The multiplicities of the references are less constrained regarding the lower bound.
- `AlterableElement` was added.

The use of the additional indirection from `NonTerminalNotation` to `Production-Part` via `Production` decouples the `Production`. The reason for this is the use of `Production` as the complementing element for a language element. This is described in the following section 4.5.4 in more detail and is one of the main design targets of

the Notation Model.
AlternativeNotation is not needed to represent a Parse Tree. It is the only ProductionPart which does not contain and does not reference to data or another ProductionPart. TerminalNotation's link to data is described in the separate diagram 4.11. AlternativeNotation's only use is to hint while the Parse Tree is constructed. For example, without ProductionPart for the given rule

```
R : ((a | b) | (c d))
```

it is possible to hint to each element individually, but it would be impossible to hint to the alternative (a | b).
To be able to hint to a specific grammar element without requiring a representation is the reason the lower bounds of all references have been removed. This must be possible during Parse Tree construction.
AlterableElement is the base class of all production related classes. Its main task is to hold alternativeRepresentations for itself. These alternativeRepresentations are not persisted and are derived by the Parse Tree Constructor or in other words set by it. AlterableElements also contains a set of persistedRepresentations maintained from former representations, for example. These do not need to be a subset of the current alternativeRepresentations. Its dedicated use is to restore former representations, which are or became again valid.

### 4.5.4 Language connection part of the Notation Model

Figure 4.10 shows how the connection of the Notation Metamodel to the language model is made. The two central metaclasses in this diagram are EObjectProduction and LanguageTokenContent. The root of the Notation Model is NotationModelRoot. It contains the rootProduction of the Parse Tree. This might be an UnassignedRuleProduction, but in general an EObjectProduction. The connection between both is established by the reference languageElement of EObjectProduction. The type of the referenced instance must be the referenced type of the EObjectTypeRule at type references of EObjectProduction. EObjectProductions are the only notation elements which establish a direct connection between the Notation Model and the language model. EObjects, like Java objects, are the only referable instance at runtime. Thus, a structurally identical containment relationship based tree with language objects on the one hand and EObjectProductions are created on the other side. The subtrees of a notation node, EObjectProduction, are stored

in its `children` containment. This complements the vanished containment relation between `NonTerminalNotation` and `Production` in figure 4.8. This design allows each `EObject` production to contain its own part of the Parse Tree, integrating Parse Tree and structurally symmetric language tree on a per `NonTerminalNotation` base. `NonTerminals` are the only part in the grammar where `EObjects` are produced, but `NonTerminals` may also produce `EDataTypes` or reference to a rule which is an unassigned rule call. To store productions in the Parse Tree where no `EObject` is produced, `NonTerminalNotation` contains a derived `nonEObjectProduction` containment. This containment is derived, because it is just set in the case where `production` is not of type `EObjectProduction`. This reestablishes the missing containment reference in the case where the `Production` is not contained by the `EObjectProduction` tree.
The other notation class, which needs to establish a connection to the language elements, is `LanguageTokenContent`. The `TokenContentAdapter` is responsible to fill in the token value while the token stream is created during token stream creation of the Parse Tree. This is explained in section 4.5.5. `LanguageTokenContent` is responsible to assign the token content based on a value of an instance of an `EStructrualFeature` of an `EObject` in the language model. Because instances of `EStructuralFeatures` are not referable, `LanguageTokenContent` uses a reference to the `EObject` and reflection to obtain the value of the `EStructuralFeature` instance defined by the `eStructuralFeature` reference of `FeatureOperation` of the attributed grammar Metamodel. The reference to the necessary language `EObject` is obtained by the first parent `EObjectProduction` in the Notation Models containment hierarchy. The `index` of `LanguageTokenContent` determines the exact value of an `eStructuralFeature` in case of a multiplicity higher than one, for example, if an `EAttribute` is the type of multiple (list of) strings.

### 4.5.5 Token value connection

Diagram 4.11 shows the relation of notation elements to the token value. Except for the `token` reference of `TokenContentAdapters`, this is only relevant for token stream creation of the Parse Tree. The abstract `TokenContentAdapter` is the central class in the diagram. Each `TerminalNotation` and each `DataTypeProduction` refers to exactly one `TokenContentAdapter`. Because both hold the content of a structureless value, this unifies both in regard to token stream creation. The `TokenContentAdapter` has a derived attribute named `tokenValue` of type string. The value of `tokenValue`

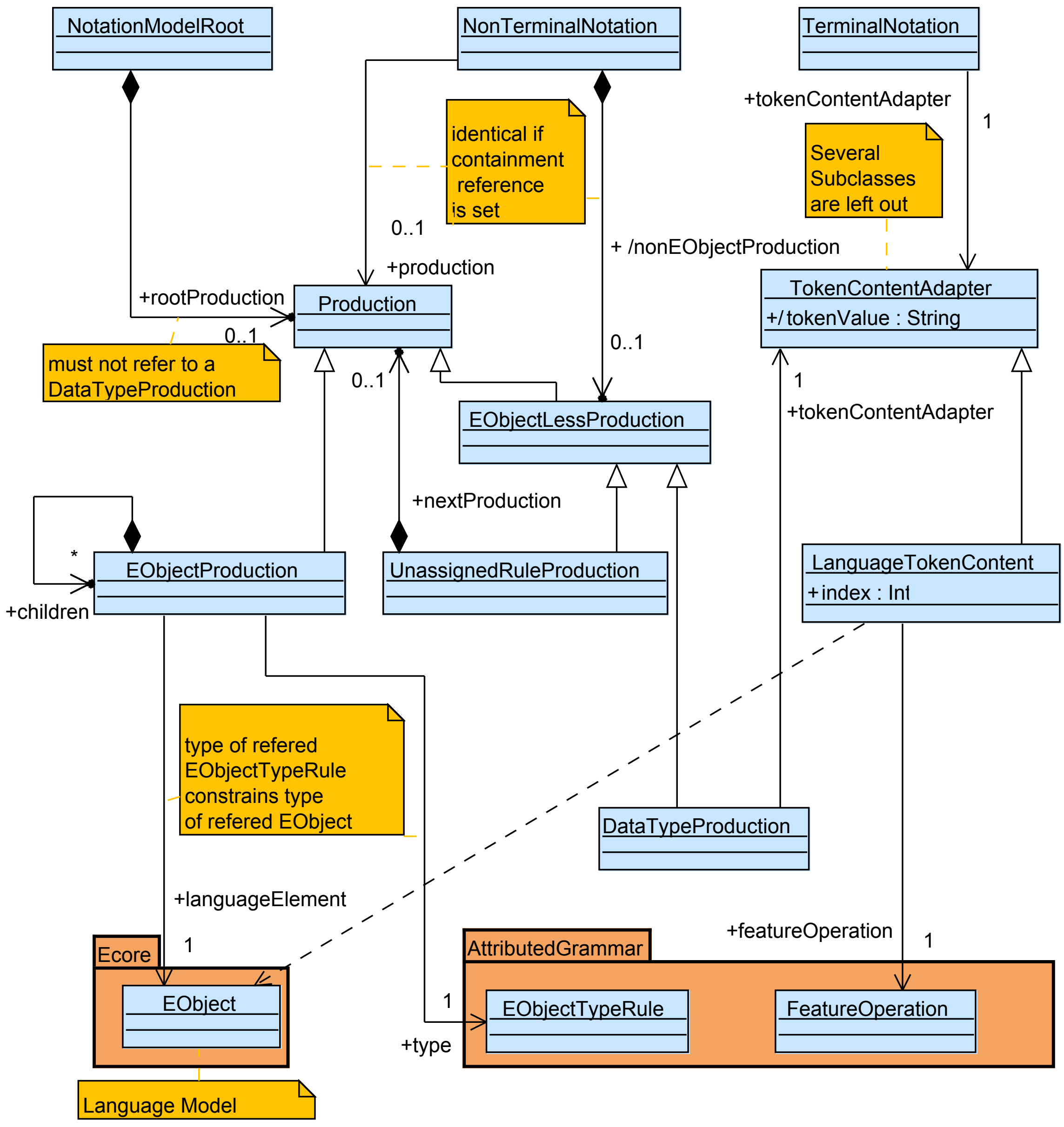

Figure 4.10: Language model connection

is determined by the use of `ValueConverter`s, in order to convert between a non-character type, for example `EInteger` and the only valid output value of Tokens, characters. The use of the transient optional reference to `Token` is twofold, first it enables the Notation Model to hold references to both, the content of a `Token` on the token stream and its content in the language model, thus optionally keeping both synchronized. Second, this existence is mandatory to hold the textual representation of cross references until they are resolved to an URI or completely. Subclasses of `TokenContentAdapter`s represent the different possibilities of data sources for the token content:

- `LanguageTokenContent` is explained in the Language connection part of the Notation Model, section 4.5.4.
- `GrammarTokenContent` the value of the token depends on a `Literal value` defined in the grammar Metamodel.
- `NotationTokenContent` the value of the token is contained by a `NotationProperty` in the Notation Model. The key is obtained by a reference to an instance of `NotationPersistedElement`s from the attributed grammar Metamodel.
- `MagicTokenContent` is used when the data source is none of the above. It is used if the content is set programmatically.

## 4.5.6 Compression of the Notation Model

To create a token stream it is necessary to create the complete Parse Tree structure in the Notation Model. This structure is more verbose than what is necessary to uniquely identify a word. Due to the fact that the Notation Model serves as a building plan to create a word, it is just necessary to persist the part of the Notation Model which eliminates all ambiguities. For example, for the following rule:

```
A:   "keyword1" v=ID
 |   "keyword2" v=ID
```

it is just necessary to store which rule was chosen, for example, the one containing `"keyword1"`, but not the expanded Parse Tree. If Parse Tree construction is predictable, it is even possible to omit this distinction if the selected option equals the default option of the Parse Tree Constructor.

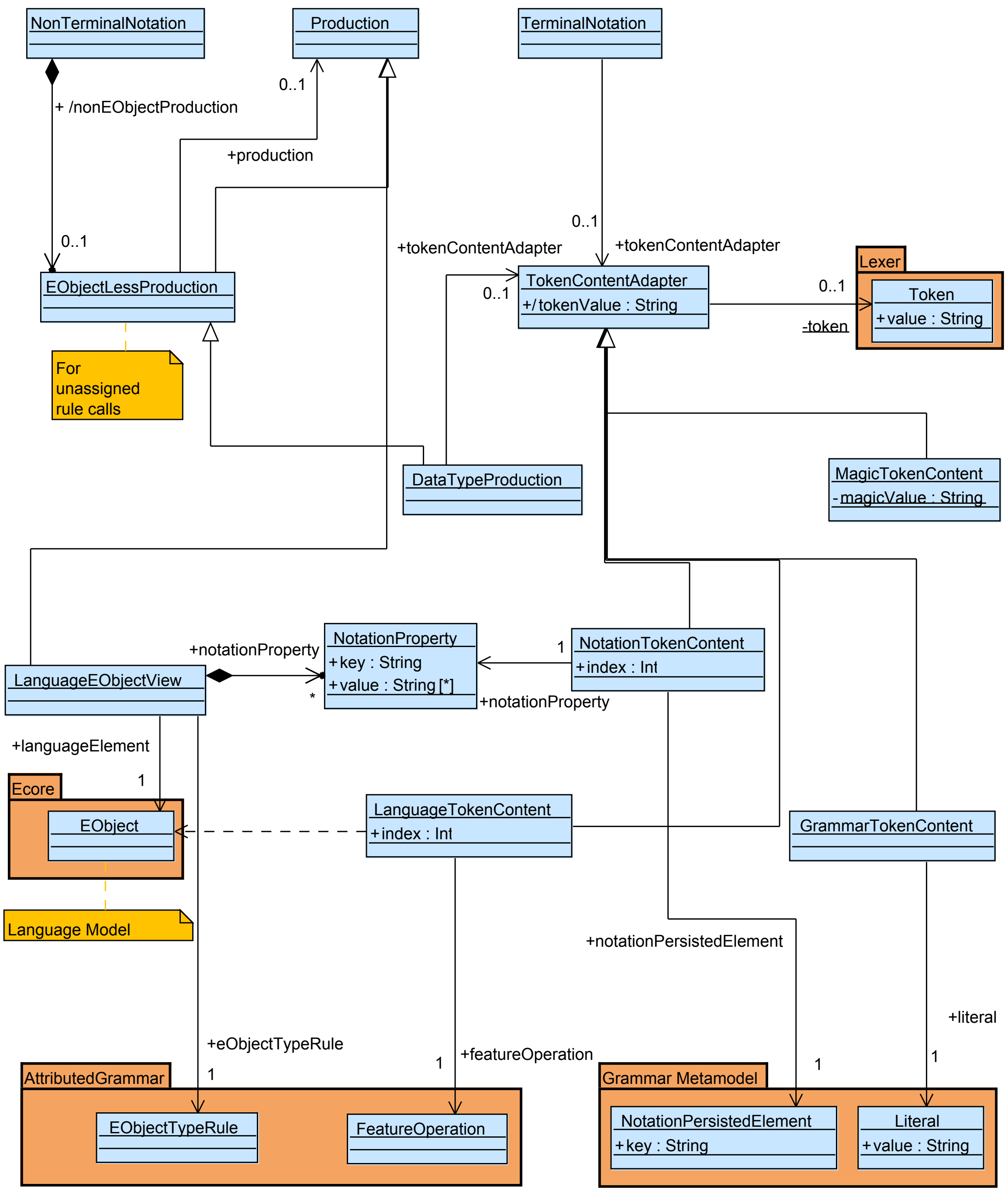

Figure 4.11: Token value connection

It is furthermore possible to enumerate the alternatives and save their values instead of the references to the `ProductionParts`.

### 4.5.7 Parse Tree Constructor Conclusion

The Guided Parse Tree Constructor combined with the Notation Model allows for the choice of different textual representations for a language model.

## 4.6 Structured Characters

This chapter presents a concept to embed structured data in an atomic character. It enables the use of structured data, like `EObjects` on the character input stream for the Lexer. In combination with typing rules, serialized Notation Model elements resolve to sentential forms of a word. The general idea is to use private characters as keys for a map. These special characters, which identify an object, are called *ID Characters* in the following.

### 4.6.1 General Requirements

The task of an ID Character is to uniquely identify an object in a serializable context. The ideal characteristics are:

- *Bindable* and existence dependent on another character or standalone.
- *Unlimited* number of unique identifiers.
- *Atomic.* The integrity of the identification should be preserved and not be breakable.
- *Standardized.* There should be a long term guarantee that they're available for use.
- *Ubiquitously* available. They should be available on every platform.
- *Abstract* characters. They should be quickly distinguishable from non-ID characters.
- *Private.* They should not cause any conflicts or be missinterpretable in any way.

#### Unicode Private-Use Characters

Unicode [JDA12] offers a possible solution. Unicode is a universal character encoding standard for consistent encoding and exchange of text data. It is the default encoding of HTML and XML and is implemented in all modern operating systems. It specifies a numeric value (code point) and name for each character. Unicode was developed in conjunction with the Universal Character Set and can be represented by widely available encodings like UTF-8. The Unicode Standard defines private-use characters, in which interpretation is not specified and is determined by a private agreement among cooperative users. For example, Apple uses a private character to present its apple logo. An application specific changed interpretation of, for example the character representing the apple logo is valid, because it specifies its intended behavior according to a private agreement. The private use characters are intended for software developers. Compared to the ideal characteristics:

- They are stand alone characters and are not bindable per se.
- The number of identifiers is *limited* to 137,468, in which 6,400 are in the private use area U+E000 to U+F8FF and 65,534 are in each supplementary private use Area-A and Area-B.
- They are single characters and thus *atomic*.
- Unicode is *standardized and private-use* characters are permanently designated for private use.
- Unicode, especially the UTF-8 encoding, is wide spread and nearly *ubiquitously* available on modern personal computer platforms.
- There are three code-blocks for private-use characters. A three range check for a code point is sufficient to determine its private-use.
- Private-use characters are, as the name states, explicitly designed for *private* use.

The restriction of a limited number of available identifiers could be solved by implementing a non-standard character encoding, such as a variable-width encoding.

### 4.6.2 Use and implementation of ID-Characters

The ID character can be to

- add an ID to another token.
- use the ID to reference to an object.

In both cases, additional data must be preserved. Because the character stream is used for serialization, the additional data should also be serialized in an adjacent file. Figure 4.12 sketches an EMF based lexer extension. The adjacent file is named `EMF Document` on the right, whereas the token stream is read from the `Text Document` at the bottom. The wrapping lexer is named `EMF Lexer` in figure 4.12, whereas the wrapped lexer is called `Original Lexer`.

**Adding an Identifier** To add an identifier to the following or previous token is useful in regards to enabling stable references and enhancing compare based updated approaches. This identifier providing use is *not of interest for this thesis.* An associativity flag is required for right, none or left associativity. Right associativity is easy to implement, because the wrapping Lexer can distinguish and filter the ID character, resolve the identifier and append it to the token value of the token returned by the wrapped lexer. If left associativity may occur, this requires the wrapping lexer to additionally cache one token of the wrapped Lexer and request another token from the wrapped lexer. Aside from being easily distinguishable from the normal textual representation, ID characters clutter the output and should not be considered human readable without preprocessing.

**Referring character** Using the ID characters to reference to objects conceptually enables structured data on the character stream. This even exceeds the complexity of normal token values, which contain a sequence of characters in general. Restricting the referenced objects to be serializable would be sufficient, but restricting the references to `EObject` eliminates impedance mismatch for a character to `EObject` mapping and also ensures a non-binary XMI serialization. Figure 4.12 realizes this by a map from UTF8 character to a triple, $c \rightarrow (o, a, i)$, where $o$ is the referenced object, $a$ is the associativity and $i$ is the extrinsic identifier. The associativity determines if it's a plain ID or a reference character. `None` associative characters are EObject references. $o$ and $i$ can be tree structures, because if $o$ contains other `EObjects`, multiple UUIDs adjacent to each `EObject` must be persisted in $i$. References between the serialized `EObjects` are by the use of UUIDs stable, but stability of references to textually described `EObjects` depend on the quality of the resource implementation and are problematic.
The workflow of an `EMF Lexer` is thus far:

1. Load the adjacent document and make the keys available.
2. wait for a next token request.
3. forward this request to the `Original Lexer` and pass the regular characters on the input stream through to it.

4. if the Lexer produces a token, return it
5. if the UTF8 private use character occurs, let the `EMF Lexer` handle it:
   - resolve the triple and the according `EObject`
   - save the UUID, either at the token by a value, or as a reference on the `EObject` or as an adapter attached to the `EObject`
6. Test if specialization `Typer Rules` exist, and if it does not assume the grammar explicitly define this type and assign the classname of the `EObject` to the token name. Finally, assign the resolved `EObject` to the token's `value`.

**Referring Sentential Forms** The assignment of the token's name is static in the previous paragraph. To leverage the approach of letting the lexical analyzer return `EObject`s, the token name can be determined by rules. These rules can be added to, and interpreted by the `EObject Typer`. The `Typer Rules` process the input `EObject` and assign the token name based on runtime analysis of the input. Sentential forms are serialized using the Notation Model. To enable the reference to sentential forms, an `EObject Typer` rule is necessary which assigns the token name properly.
The workflow of the previous paragraph 4.6.2 is extended by a `Typer Rule`, which:

1. Tests the input `EObject` if it is an instance of `AlterableElement`. If it is not, the rule is not applied.
2. Tests if the input is an instance of `NotationModel:List`. If it is not a `List`:
   a) Resolves the grammar element by following the `notTerminal`, `sequence` or `terminal` reference.
   b) For `Symbols`, resolves their more generic referenced `TerminalType` or `Rule`.
   c) Determines their URI and assigns it to the token name.
   d) Assigns the referenced language element `EObject` to the tokens value.
   e) Optionally validates if the language element structure complies to all constraints for the resolved grammar element.
3. If the current `EObject` is a `List`, the `EMF Lexer` returns multiple tokens for a single character:

- Removes the `AlterableElement` at the head of the list and resolves it like a stand alone element in the above steps.
- Keeps the smaller `List` on the input stream or redirects the next `nextToken()` from the Lexer directly to the current `Typer Rule`.

The set of terminals is composed of:
$Terminals = Terminals_{OriginalLexer} \bigcup Terminals_{EMFObjects} \bigcup Terminals_{Sentential}$

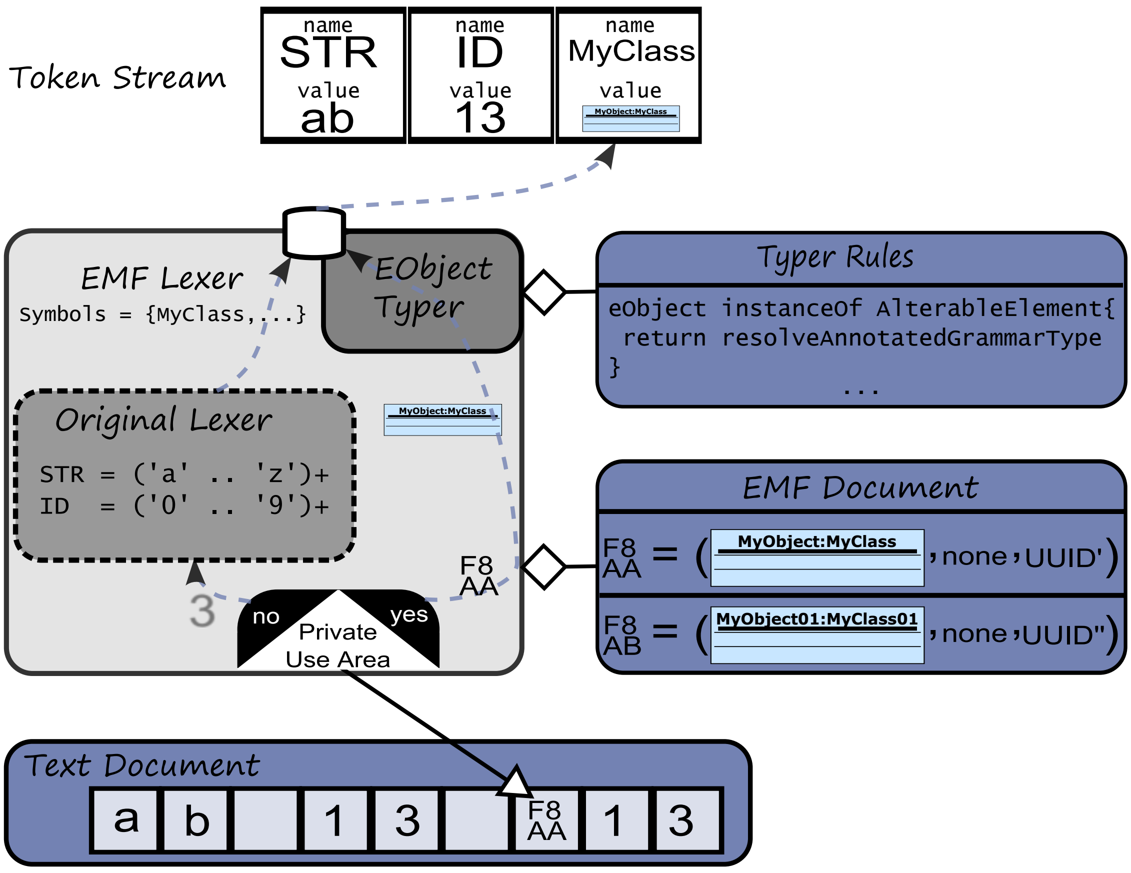

Figure 4.12: EMF Lexer extension

## 4.7 Integrated Use

The previous chapters demonstrated the Guided Parse Tree Constructor, the Notation Metamodel and the EMF Lexer as separated as possible. This chapter focuses on their integrated use and their synergy effects. It start with the use of the EMF Lexer to allow the declaration instead of the description of the whole language model, continues with combined use of Parse Tree Constructor and Notation Model to allow different semantically equivalent representations. The implementation of Sentential Tokens is then addressed which leads to Sentential Tokens and Graphical Editors.

It is assumed that:

1. The language model stays stable if its textual representations are unchanged and that the nodes are reused to a large extend.
2. The textual representation is additional to the abstract representation, so that non-containment references remain stable and non-`EObject` referencing, but `EObject` dependent sentential tokens remain valid.

**Incomplete information handling** The first, obvious benefit of the `EMF Lexer` is that the textual representation does not necessarily need to *describe* the whole model, but just needs to *declare* it. It enables the use of `EObjects` directly in the grammar and thus the grammar to properly handle parts of the language. This results in the inability to modify the unhandled parts and the existence of an ID character, or a character placeholder at the unhandled location. This defers the problem of handling this placeholder to the textual editor. Unaware editors represent UTF8 private use characters, either as their canonical representation with a square filled with the value of the code point in a hexadecimal representation or simply as a square □.

**Different representations of semantic equivalent tokens** In case the Parse Tree Constructor found more than one valid grammar rule for a notation element, the corresponding transient field is set. The text editor should offer the user to change the current representation, which reassigns the pointer to the grammar rule of the indirectly selected notation element and, if necessary, also to its descendants. As soon as the reassignment is finished, a new token stream is produced reflecting the new textual representation.

**Sentential Tokens** A Sentential form is an intermediate form for creating words. Tokens which store a part of the Parse Tree are called *Sentential Tokens* in this thesis. The name originates from the idea to store the tree structure of produced non-terminal $N$ in a token $n_S$ and let the parser use them interchangeably. In this thesis, this concept is broadened to `Terminal`s and `Sequence`s. The concept of sentential forms is necessary for incremental parsers and can be leveraged for visual editing.

The EMF Lexer section 5.1 describes the possibility to serialize Notation Model elements and assign the token name according to the referenced element of the grammar model, but lacks further parser integration. The suggested approach is not to add special functionality to the parser framework, but to post process the grammar before it's used for the parser framework. If the parser framework uses a generative approach, either code or parse table generation, this post processing is transparent for the language designer and does not require additional effort to them. Post processing of input grammar requires the following steps:

1. collect all `Rule`s, `TerminalType`s and `Sequence`s of the grammar in a set $\mathbb{T}$.
2. for each $t \in \mathbb{T}$ create a new terminal $t_S \in Terminals_{Sentential}$. $Terminals_{Sentential}$ are part of $Terminals$ created by the `EMF Lexer`, see 4.6.2, and are added to the `Terminal` list of the grammar.
3. collect all references to each `Rule` and to each `TerminalType` together with all `Sequence`s in $\mathbb{S}$.
4. replace every $s \in \mathbb{S}$ with $(s|t_S)$.

The above steps replace every occurrence of a `Sequence` or `Symbol` with an alternative with itself or its sentential terminal $t_S$. For example:
A : a | B
is transformed to
A : (a | a$_S$) | (B | B$_S$)

This procedure allows the deserialization of sentential forms of the language from the character stream. The `visible` flag of the production part indicates if the notation element of a symbol should be serialized as an ID character or not. It is not necessary to create the Parse Tree equivalent nodes contained by an invisible flagged ID character. The invisible flagged ID character guarantees that it is possible to create a valid production of the hinted unresolved production together with the referencing language `EObject` and its directly and indirectly contained `EObject`s.

**Sentential Tokens and Graphical Editors** Sentential tokens are the foundation for mixing graphical editors into text. Sentential Tokens alone, as described in 4.7, are, without leverage, just a way to fold text into a special character. For graphical editors, they provide data access and serve as place holders. Sentential tokens restrict the underlying model to conform to certain criteria. To provide alternative graphical editors as an alternative to textual parts, an integration at the same place like alternative textual presentations is desirable.

In order to enable graphical editors that conform to a context free grammar, they have to be a specialization of an invisible instance of a grammar part. This enables their integration to the Notation Model and allows generic handling regarding Lexer, parser and Parse Tree. Graphical editor integration in the Notation Metamodel 4.7 is based on specialization of one or multiple subclasses of `ProductionPart`s. Multiple subclassing is, for example, useful if the editor and is able to represent one or a `List` of specific `NonTerminalNotation`. This subclass is denominated as the *graphical class* and its serialized form as the *graphical character*. From the parsers perspective, this enables the use of the graphical character wherever the specialized `ProductionPart` is valid. This concludes that the graphical class is only a valid path for the Parse Tree Constructor if the specialized `ProductionPart` is also valid.

The graphical editor has to specify which grammar element or set of grammar elements it is able to represent. Because the graphical class is a specialization, it is perfectly valid to further constraint the extended element, for example, to meet certain domain specific criteria. If the criteria are met, the graphical class should be added as an alternative to, not instead of, the Notation Model. The designated place to evaluate these constraints is the Parse Tree constructor, after it has determined the extended grammar element as a valid path.

The combined use of Sentential Characters and the Guided Parse Tree Constructor enable the creation of graphical characters for the output character stream. The intended representation of graphical characters in the user interface is to substitute the graphical character with its adjacent graphical editor. The graphical editor at this position *works on the language model* referenced by the graphical character. Editing the language model instead of the verbose Parse Tree is only possible due to the guarantee of sentential tokens to be expandable to a valid Parse Tree. Handling graphical elements as characters is known even by end users. Office software, such as Libre Office [2], offers to place the anchor of a picture in a document as a character.

The presented solution offers to create graphical editors as an alternative to parts of a textual representation. This allows the editors to be fine grained for specific parts of the model. There is no need to handle the model root or to query specific model parts. It allows multiple graphical editors from different sources to coexist without interference in a single textual document using an attributed context free grammar as the common denominator. Changing the representation is achieved without a change in the language model and by simply changing a reference in Notation Model. Representing text in the graphical editors can be achieved by spawning a new parser instance inside the graphical editor with an appropriate synchronized new root. Thus, interchangeable mixed representation of textual and graphical notation is possible.

[2] http://www.libreoffice.org/

# 5 Discussion

In this chapter the advantages and disadvantages of the developed EMF Lexer, Notation Model, Parse Tree Constructor and Sentential Characters are discussed. Because no EMF text framework provides stable models, the effect of unstable models on each individual result is explained.

## 5.1 EMF Lexer

The presented Lexer extension of an referencing character in section 4.6.2 allows the use of arbitrary serializable objects on the character stream as atomic UTF8 characters. The serializable objects are stored in an adjacent file as values of a map from UTF8 character keys. This file must be delivered and localized together with the original character stream. The lexer extension allows the textual language to handle complex structured data without describing it, but still requires a declaration of their existence at a specific location. This makes it impossible for the user to change the declared data. The use of private use UTF8 characters clutters the output text file by the presence of squares, □. The meaning of the squares is completely intransparent for the user, which makes them a high potential error source. If mandatory, their absence invalidates the document and makes it nearly impossible for the user using a normal text editor to return to a valid state. Creating them with a text editor is impossible and copying them from the adjacent document is unacceptable because the meaning of the value in the key value pair where it is copied from is not obvious at best. Private use characters offer a remarkably simple and, to the current knowledge, the only solution to virtually enable deep structured atomic data on the character stream. The disadvantages are severe. Their use outside a particular adapted environment should be avoided. Private characters at the start and the end of a document can be an acceptable exception.

## 5.2 Notation Model

The Notation Model is the central component for interchangeable representations of the same language element and is especially sensitive to an unstable language model. The concept of a Notation Model already proved valuable for graphical editors like GMF and also in the reduced layout oriented version by XText. The drawbacks of the Notation Model is its consumption of resources and its dependency on the stability of the language model. The increased resource consumption of the Notation Model is due to the increased memory consumption of the Notation Model itself. For an uncompressed model, this results in at least one notation element per token. Keeping old states and providing alternative presentations intensifies memory consumption. Providing a stable language model is the most challenging prerequisite. In stable language models, elements are reused and only the elements for which their textual representation are altered are changed. This is not the case for current EMF based textual editing tools. [Wag98] offers an incremental solution which reuses nodes to a large extent. Barista[1] adopted and simplified [Wag98]'s algorithm to work on abstract syntax trees [KM06]. It remains open as to whether this adoption is also possible for the parse to abstract syntax tree conversions used by Xtext or in regards to the simplified version of this thesis. Tree rewrites are necessary in Xtext's grammar because of characteristics of the underlying LL(*) parsing algorithm, but not for Generalized Left-to-right Rightmost derivation parsers, for which [Wag98] offers an incremental solution and were therefore excluded in the grammar model of this thesis. If the language model is not stable, recovering the Notation Model by heuristically

[1] http://www.cs.cmu.edu/~NatProg/barista.html

comparing the old with the new language model is possible, but not regarded further in this thesis. The EMF Compare[2] Framework provides a customizable `MatchService` to facilitate this approach.
It is possible to recreate the Notation Model on each parse based on the notation information which can be deduced from the *current* textual representation. This is the approach used by Xtext for its Node Model. For the described Notation Model, this leads to the loss of not deduceable information, like `persistedRepresentations` and the need to recalculate `alternativeRepresentations`, as well as *non-identical* referenced `EObjects` of the language model. In contrast to Xtext, it is important to keep the presented Notation Model up to date even after pretty printing. The Notation Model offers the ability to switch between different representations and must be only invalidated by the first textual change. The non-identical references are *the* common problem of unstable models, because a new language model is created and some parts might still reference to the old model. Converting the references to proxies containing an URI does not ensure the stability of the referenced target. This problem occurred for Xtext only for cross-document references; this problem also occurs in a single document if using ID characters. In detail, this problem occurs only if elements of the `EObject` graph reference to textual representations, not to other `EObjects` held by ID characters.
The Notation Model offers the possibility to switch between different textual presentations using the Parse Tree Constructor or provides the base for sentential characters. The problems regarding the Notation Model are the increased memory consumption and the fact that the non-Notation Model inflicts problems of unstable language models.

## 5.3 Parse Tree Constructor

The presented Parse Tree Constructor uses Xtext and TEF-like backtracking, which has a runtime of $O(c^N)$. Xtext tries the last valid option first and just requires one valid solution. The requirements for the Parse Tree Constructor for alternative textual representations requires *all* valid solutions. This exponentially increases the average runtime, for example, for the rule

```
R : (v += NUM | x += STR)*
```

[2]http://www.eclipse.org/emf/compare/

, $(|v|-1)*(|x|-1)$ are valid solutions. If similar rules exist in the grammar, the Parse Tree Constructor suffers from combinatorial explosion even for small examples. Alternative textual representations should be calculated *on demand* for a selected element. The time complexity of the Parse Tree Constructor leaves much space for improvement. It remains open for optimizations of GLR parsers, for example, sharing common prefixes and suffixes are possible for Parse Tree construction on a linearized abstract syntax tree. The Parse Tree Constructor suggested in this thesis is similar to Xtext's Parse Tree Constructor except:

- It does not construct trees for valid invisible notation elements.
- It determines all valid alternatives.
- The order in which possible solutions are tried can be partly customized. Selected alternatives, so called hints, must be tried first.
- It should be extensible to add additional specialized alternatives to the Notation Model based on optional constraints.
- It must consider different associativity if a generalized parser is used.

## 5.4 Sentential Characters

Sentential characters are virtually able to contain parts of the Parse Tree on the characters stream due to the use of the EMF Lexer and the Notation Model. Preventing the Parse Tree Constructor to construct the Parse Tree for designated Sentential characters requires that the referenced language `EObject` and its directly and indirectly contained `EObjects` are serialized together with the Notation Model element. By postponing or avoiding the Parse Tree construction, the elements of the language model are kept in their compact and abstract form. The serialized Notation Model elements designate the omitted production and thus enable the Parse Tree Constructor together with the referenced `EObject` to make the postponed Parse Tree construction up, without the need of any further information. Due to the extensibility of the Parse Tree constructor, specialized types matching can be created which match additional constraints.
Specialized types are designed to provide in conjunction with graphical editors, which use the position of the sentential characters for representation as well as their referenced language elements as data source. It is remarkable that sentential character

based editors would be parts of a valid word that is conformed to a context free grammar. There would be *no* impedance mismatch integrating sentential characters based on graphical editors in an existing text. The design time of the graphical editor is decoupled from the grammar or parser. This allows for example library designers to provide a domain specific graphical editor for their library use, which is determined by constraints, *after* the textual language is shipped. This concept also allows various editor extensions of different vendors without mutual knowledge.

There are a number of concept related restrictions. The location of the sentential character is fixed, thus, if the editor does not substitute the character at its location, the character must be there and handled. If this character is just hidden, it disables the user to recognize certain problems. Accidental deletion of the character is another discouraging example of the different behavior of the described model depending on if an element is before or after the invisible character. The integration points are fixed to certain grammar language elements. For example, a `Sequence` can be substituted, but not if it is an arbitrary sequence, for example, `b` and `c` *together* in the following example

```
R : a b c d
```

To handle more generic cases than `b` and `c`, an additional matching or specialization phase has to be introduced. The graphical editor would furthermore obtain multiple roots with multiple presentation locations. Handling multiple placeholders in a suitable representation is the responsibility of the user interface, and for the case of `b` and `c` merging is conclusive, but for arbitrary cases, such as substituting `b` and `d`, this is demanding.

The existence of the editor at a textual location is only a valid possibility if it substitutes are sentential characters. The sentential character is valid if its referenced language model satisfies certain constraints. If editing operations of the graphical editor on the language model violate these constraints, the editor renders itself invalid at the current position and seizes to exist. Because these constraints for the textual representation are added as validation constraints for the language metamodel, violation is noticed by graphical editors when the model is validated. Restricting the user to make only valid editing operations would be obtrusive and intransparent to the user. It is also common for users using editors, for the transition from one valid to anther valid state to contain invalid states. A possible solution for user interface frameworks, which provide graphical editors which possibly violate the validation

constraints for the sentential character they represent, is to just update the language model if these constraints hold and otherwise lock the language model for other editing operations completely. Another more sophisticated possibility would be based on EMF Model Transactions[3] for the transition between valid states.
The effects of unstable language models on Sentential Characters are less intense than for the Notation Model. This is due to the characteristics that the identity of referenced language objects is identified by the Sentential Character. In the case where the textual representation solely describes whole models, and the Notation Model is rebuilt after each reparse, the Notation Model element which is serialized as a sentential character must contain the referenced language `EObjects`. Sentential Characters represent the production part, so there is no overlap with other `EObject-Productions`. Because `ProductionParts` contained in `EObjectProductions` depend on the referenced language element, editing operations would change the attributed `EObject` of the reference holding, outer `EObjectProduction`. This means that parts of a single language `EObject` are edited by text and the parts by a graphical editor. For unstable language models, it is possible that the graphical editor changes certain properties of the `EObject` where the complete `EObject` is then replaced due to changes in the textual version. Therefore, Sentential Characters *must be restricted* to `EObjectProductions` if the language model is unstable. The other restriction of Sentential Characters, which they share with ID characters in general, is reference stability. A possible solution would be to use the same intermediate form for resolving references as XText does. Cross-references of serialized `EObjects` would be annotated with the textual description of the referenced target. The textual editor must update the annotated textual description accordingly, because the user is incapable of doing so.
An advantage over independent textual and graphical editors is that changed `EObjects` in the graphical editor do not require an update of the textual representation.
A conceptional disadvantage of the presented Sentential Token integration by post processing the grammar is the increased size of the parser table, due to replacing every non-alternative occurrence of $s \in \mathbb{S}$ on the right hand side of a production with $(s|t_S)$.
The presented approach of fine grained graphical editors might lead to more, potential functional overlapping editors. For example, if a graphical editor for `B` and `A` should be provided in the following grammar, this leads to two different graphical editors, where one for `B` is likely to be a subset of the one for `A`:

[3]http://www.eclipse.org/modeling/emf/?project=transaction

```
A : B c
B : b
X : B+
```

While the concept of Sentential Characters is not specific to certain graphical editor frameworks, a practical solution for graphical editors would require a specific runtime environment. The main benefit of Sentential Characters, their use as the base for graphical editors, depends on the dependencies of these editors. Without further abstraction from the runtime environment, this practically results in an *integrated development environment lock*.

# 6 Conclusion

**Alternative Textual Representations** By conceptually altering the existing Parse Tree Constructor of Xtext, different textual representations are possible. The possibly ambiguous mapping from model to Parse Tree is leveraged to allow different, context free grammar conform representations. For a user, a high degree of ambiguity is desirable, because it provides them with a higher variety of semantically equivalent representations. The extension of the Parse Tree Constructor requires the calculation of all valid solutions instead of one valid solution. Considering that the current Parse Tree Constructor uses Backtracking, it is not suited for a high degree of ambiguity. Plain Backtracking does not scale into calculating all solutions, especially regarding the combinatorial explosion of alternative combined list elements; for practical use an improved algorithm is required. The alternative textual representations are different Parse Trees, thus the representations do not just alter token values, but also *alter the structure*. This can be leveraged during grammar design time to avoid manual model transformations.

**Structured Characters** The use of EMF Lexers Structured Characters enables `EObjects` on the character stream *as Terminals*. This eases the constraint that the grammar must *describe* all model elements, but still requires that their occurrence is *declared* in it. This allows the user to textually change or create those Structured Terminals, so if the model should ever be editable outside a Structured Character aware editor, their use is discouraged. Unhandled Structured Characters are not usable, because they are a "black" box (□) from the user's perspective.

**Sentential Character based Graphical Editors** Sentential forms of the Parse Tree can be persisted and used instead of the original input, [Wag98]. Post processing of the grammar to contain Structured Characters instead of non-terminals allows the use of sentential forms on the character stream, but results in large Parse Tables. A sentential node is the root of the persisted subtree and represents a production part in the grammar. In conjunction with the corresponding language model elements, the Parse Tree Constructor can produce a valid sentential form at any time, which makes actual parse subtree construction for sentential characters obsolete. The Sentential Character is valid if the corresponding language model elements conform to the constraints of its production part. While displaying the character stream, a graphical editor can substitute the Sentential Character and operate on its model elements as long as changes keep the model elements in a state that conforms to the production part constraints. This procedure allows the integration of graphical editors into texts which still *conform to a context free grammar.* During context free grammar definition time, it is not required to consider the existence of graphical editors. Graphical editors can substitute *any* symbol and `Sequence`, which enables fine grained editors, but with potential overlapping functionality. To determine a specific graphical editor for a sentential character, the referenced `EObject` is specialized by the Parse Tree Constructor. Thus, incremental visual domain specific language development from an internal textual domain specific language is possible. The graphical editor extensions of the language are locked to the editor framework. Because language extensions in general are locked to the language Metamodel, a common Metamodel for a textual language should be shared across the editor framework.

**Effect of Unstable Models** If language model elements are replaced instead of updated for textual changes, the language model is called *unstable.* Referential integrity for existing references or URI based references can not be kept. Parsing technique used by the EMF related textual editing tools like EMFText, Xtext and

Textual Editing Framework results in unstable models, therefore the effect of unstable models to the presented solutions is important.

- The Notation model has to be rebuilt on every textual update. Thus, every information which can not be derived from the textual representation is lost.
- References of `EObjects` in Structured Characters that reference `EObjects` which might get updated due to textual changes are unstable. This can be circumvented by erasing the reference and storing a textual identification in an `Annotation` to resolve the reference like any textual reference. Changes to the identification of the referenced target, must be taken over by the editor to the textual identification in the `Annotation`.
- Sentential Characters must be handled, because Parse Tree construction was omitted due to the existence of the Parse Tree Constructor and the referencing `EObject`. To avoid merging of old and new versions of the language model by using the Notation model, a containment reference instead of a cross reference from the `EObjectProduction` to the referenced language `EObject` is used. Notation model elements depend on the data of their containing `EObjectProduction`, if they are none themselves. Therefore, Sentential Characters are restricted to `EObjectProduction`, or to parts of the attributed grammar that produce or refer to an `EObject` themselves.

**EMF based Textual Frameworks** At the current state EMF based textual frameworks produce only unstable models. Thus cross-references created by graphical editors must be *manually maintained each time the textual representation is changed.* This work is error prone and not maintainable for large, frequently changing models. Where no intrinsic identification of language elements exist in the language, manually set cross references are unusable. Cross references are one of the key advantages of languages defined by Metamodels over context free grammars. The problem of unstable models is caused by the use of certain existing Parser generators like ANTLR for Xtext and EMFText and RunCC [1] for TEF. These Parser generators are designed to support the creation of a language based on a context free grammar. Without cross references, two equal Parse Trees are interchangeable. In practice, created references like symbol tables, are automatically derived from the textual representation which solely describe the abstract syntax tree, so the problem does not become eminent.

[1] http://runcc.sourceforge.net/

However, parser technologies were developed that optimally reuse existing parse trees to avoid CPU intensive post processing. Reusing the Parse Tree allows manually set cross-references. The choice of existing EMF based textual frameworks to use non-incremental Parsers for EMF integration instead of incremental Parsers like [BG06] should be reconsidered.

Despite the problem of unstable models, alternative graphical representations by grammar conform Sentential Characters remain feasible, making graphical internal domain specific languages possible. □

# Bibliography

[ALSU06] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[APCS03] Alanen, Marcus ; Porres, Ivan ; Centre, Turku ; Science, Computer: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. 2003. – Forschungsbericht

[Bac59] Backus, John W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In: *IFIP Congress*, 1959, pages 125–131

[BG06] Begel, Andrew ; Graham, Susan L.: XGLR: an algorithm for ambiguity in programming languages. In: *Sci. Comput. Program.* 61 (2006), August, Nr. 3, 211–227. `http://dx.doi.org/10.1016/j.scico.2006.04.003`. – DOI 10.1016/j.scico.2006.04.003. – ISSN 0167–6423

[BGVDV92] Ballance, Robert A. ; Graham, Susan L. ; Van De Vanter, Michael L.: The Pan language-based editing system. In: *ACM Trans. Softw. Eng. Methodol.* 1 (1992), January, Nr. 1, 95–127. `http://dx.doi.org/10.1145/125489.122804`. – DOI 10.1145/125489.122804. – ISSN 1049–331X

[BKVV08] Bravenboer, Martin ; Kalleberg, Karl T. ; Vermaas, Rob ; Visser, Eelco: Stratego/XT 0.17. A Language and Toolset for Program Transformation. In: *Science of Computer Programming* 72 (2008), June, Nr. 1-2, pages 52–70. – ISSN 0167–6423. – Special issue on experimental software and toolkits

[BLFM05] Berners-Lee, T. ; Fielding, R. ; Masinter, L.: *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. http://rfc.net/rfc3986.html. Version: 2005

[Bos01] Boshernitsan, Marat: Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools / EECS Department, University of California, Berkeley. Version: Jun 2001. http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/5819.html. 2001 (UCB/CSD-01-1149). – Forschungsbericht

[EMF] *Eclipse Modeling Framework Runtime and Tools - Eclipse Help.* : *Eclipse Modeling Framework Runtime and Tools - Eclipse Help*. Version: 2.8.0.v20120608-0554

[EMF12] *EMF Text USER GUIDE.* : *EMF Text USER GUIDE*, June 1 2012. http://svn-st.inf.tu-dresden.de/svn/reuseware/tags/current_release/EMFText/org.emftext.doc/pdf/EMFTextGuide.pdf

[Fow05] Fowler, Martin: Language Workbenches: The Killer-App for Domain Specific Languages? (2005), May. http://www.martinfowler.com/articles/languageWorkbench.html

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[GMF] *Eclipse Graphical Modeling Framework (GMF) Documentation - Eclipse Help.* : *Eclipse Graphical Modeling Framework (GMF) Documentation - Eclipse Help*. Version: 1.4.0.v201206221900

[Gro09] Gronback, Richard C.: *Eclipse modeling project : a domain-specific language toolkit*. Upper Saddle River (N.J.), Boston, Indianapolis : Addison-Wesley, 2009 (The eclipse series). http://opac.inria.fr/record=b1128309. – ISBN 0–321–53407–7

[Gru10] Grune, Dick: *Parsing Techniques: A Practical Guide*. 2nd. Springer Publishing Company, Incorporated, 2010. – ISBN 1441919015, 9781441919014

[HHKR07] Heering, J. ; Hendriks, P.R.H. ; Klint, P. ; Rekers, J.: *The Syntax Definition Formalism SDF - Reference Manual.* 2007

[JDA12] Julie D. Allen, Joe Becker Richard Cook Mark Davis Peter Edberg Michael Everson Asmus Freytag John H. Jenkins Rick McGowan Lisa Moore Eric Muller Addison Phillips Michel Suignard Ken W. Deborah Anderson A. Deborah Anderson (Hrsg.): *The Unicode Standard Version 6.1 – Core Specification.* The Unicode Consortium, 2012. – ISBN 978–1–936213–02–3

[jet12] jetbrains (Hrsg.): *MPS User's Guide.* 2.5. jetbrains, 09 2012. http://confluence.jetbrains.net/display/MPSD25/MPS+User%27s+Guide

[KAM05] Ko, Andrew J. ; Aung, Htet ; Myers, Brad A.: Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In: *Proceedings of the 27th international conference on Software engineering.* New York, NY, USA : ACM, 2005 (ICSE '05). – ISBN 1–58113–963–2. http://dx.doi.org/10.1145/1062455.1062492, 126–135

[KM05] Ko, Andrew J. ; Myers, Brad A.: Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In: *Proceedings of the 18th annual ACM symposium on User interface software and technology.* New York, NY, USA : ACM, 2005 (UIST '05). – ISBN 1–59593–271–2. http://dx.doi.org/10.1145/1095034.1095037, 3–12

[KM06] Ko, Andrew J. ; Myers, Brad A.: Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In: *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems.* New York, NY, USA : ACM Press, 2006. – ISBN 1595933727. http://dx.doi.org/10.1145/1124772.1124831, 387–396

[KV10] Kats, Lennart C. ; Visser, Eelco: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications.* New York, NY, USA : ACM, 2010

(OOPSLA '10). – ISBN 978–1–4503–0203–6. http://dx.doi.org/10.1145/1869459.1869497, 444–463

[Mav97] MAVERICK, V.: *Presentation by Tree Transformation.* University of California, Berkeley, 1997 (Report (University of California, Berkeley. Computer Science Division)). http://books.google.de/books?id=1XwvNAAACAAJ

[Moh93] MOHER, Mak D.C. Blumenthal B. Leventhal L. T.G.: *Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In: Empirical Studies of Programmers: Fifth Workshop : Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA.* Ablex Publishing Corporation, 1993 (Human/computer interaction). – 155 S. http://books.google.de/books?id=rMmxq8q0CGYC. – ISBN 9781567500899

[MPS12] ; jetbrains (Veranst.): *The calculator language tutorial.* http://www.jetbrains.com/mps/docs/tutorial.html. Version: 09 2012

[Obj10a] OBJECT MANAGEMENT GROUP: *UML 2.3 Infrastructure.* http://www.omg.org/spec/UML/2.3. Version: may 2010

[Obj10b] OBJECT MANAGEMENT GROUP: *UML 2.3 Superstructure.* http://www.omg.org/spec/UML/2.3. Version: may 2010

[OMG] http://www.omg.org/

[Par07] PARR, Terence: *The Definitive ANTLR Reference: Building Domain-Specific Languages.* First. Pragmatic Bookshelf, 2007 (Pragmatic Programmers). http://www.amazon.com/Definitive-ANTLR-Reference-Domain-Specific-Programmers/dp/0978739256%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0978739256. – ISBN 0978739256

[SBPM09] STEINBERG, David ; BUDINSKY, Frank ; PATERNOSTRO, Marcelo ; MERKS, Ed: *EMF: Eclipse Modeling Framework 2.0.* 2nd. Addison-Wesley Professional, 2009. – ISBN 0321331885

[Sch04] SCHRAGE, Martijn M.: *Proxima – a presentation-oriented editor for structured documents*, Utrecht University, The Netherlands, PhD thesis, Oct 2004. http://www.oblomov.com/Documents/Thesis.pdf

[Sch08] SCHEIDGEN, Markus: Textual Modelling Embedded into Graphical Modelling. In: *ECMDA-FA*, 2008, pages 153–168

[SS08] SCHRAGE, Martijn M. ; SWIERSTRA, S. D.: Beyond ASCII - Parsing Programs with Graphical Presentations. 14 (2008), Nr. 32, pages 3414

[SV05] STAHL, T. ; VÖLTER, M.: *Modellgetriebene Softwareentwicklung*. dpunkt-Verl., 2005 http://books.google.de/books?id=MKb6AAAACAAJ. – ISBN 9783898643108

[Vis97] VISSER, Eelco: Scannerless Generalized-LR Parsing / Programming Research Group, University of Amsterdam. 1997 (P9707). – Forschungsbericht

[Vol11] VOLTER, Markus: From Programming to Modeling - and Back Again. In: *IEEE Softw.* 28 (2011), November, Nr. 6, 20–25. http://dx.doi.org/10.1109/MS.2011.139. – DOI 10.1109/MS.2011.139. – ISSN 0740–7459

[Wag98] WAGNER, Tim A.: Practical Algorithms for Incremental Software Development Environments. Berkeley, CA, USA : University of California at Berkeley, 1998. – Forschungsbericht. – UMI Order No. GAX98-03388

[WG97] WAGNER, Tim A. ; GRAHAM, Susan L.: General Incremental Lexical Analysis. In: *I = I=16*, Unpublished, 1997

[WG98] WAGNER, Tim A. ; GRAHAM, Susan L.: Efficient and flexible incremental parsing. In: *ACM Trans. Program. Lang. Syst.* 20 (1998), September, Nr. 5, 980–1013. http://dx.doi.org/10.1145/293677.293678. – DOI 10.1145/293677.293678. – ISSN 0164–0925

[XTe12a] *Xtext 2.3 Documentation.* : *Xtext 2.3 Documentation*, June 28 2012. http://www.eclipse.org/Xtext//documentation/2.3.0/Documentation.pdf

[XTe12b] *XText 2.3 JavaDoc.* : *XText 2.3 JavaDoc*, 2012. http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.3/

# Affidavit

I hereby declare that this thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. This thesis has not been submitted to any other board of examiners in its present or a similar form and was not yet published in any other way.

Wedel, September 17th, 2012

Stefan Kuhn