HAL
open science

Model Driven analysis and synthesis of textual concrete syntax

Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schnekenburger, Sébastien Gérard, Jean-Marc Jézéquel

To cite this version:

Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schnekenburger, et al.. Model Driven analysis and synthesis of textual concrete syntax. Software and Systems Modeling, 2008, 7 (4), pp.423–442. 10.1007/s10270-008-0088-x . inria-00468231

HAL Id: inria-00468231

https://inria.hal.science/inria-00468231v1

Submitted on 1 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-Driven Analysis and Synthesis of Textual Concrete Syntax

Pierre-Alain Muller[1], Frédéric Fondement[2], Franck Fleurey[3], Michel Hassenforder[2], Rémi Schnekenburger[4], Sébastien Gérard[4], Jean-Marc Jézéquel[1]

[1] IRISA / INRIA Rennes
Rennes, France
{pierre-alain.muller, jean-marc.jezequel}@irisa.fr

[2] Université de Haute-Alsace, MIPS
Mulhouse, France
{frederic.fondement, michel.hassenforder}@uha.fr

[3] SINTEF
Oslo, Norway
franck.fleurey@sintef.no

[4] CEA, LIST
Gif-sur-Yvette, France
{remi.schneckenburger, sebastien.gerard}@cea.fr

Abstract. Meta-modeling is raising more and more interest in the field of language engineering. While this approach is now well understood for defining abstract syntaxes, formally defining textual concrete syntaxes with meta-models is still a challenge. Textual concrete syntaxes are traditionally expressed with rules, conforming to EBNF-like grammars, which can be processed by compiler compilers to generate parsers. Unfortunately, these generated parsers produce concrete syntax trees, leaving a gap with the abstract syntax defined by meta-models, and further ad-hoc hand-coding is required. In this paper we propose a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes to textual concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations.

1 Introduction

Meta-languages such as MOF [1], Ecore [2], Emfatic [3], KM3 [4] or Kermeta [5], model interchange facilities such as XMI [6] and tools such as Netbeans MDR [7] or Eclipse EMF [2] can be used for a wide range of purposes, including language engineering. While meta-modeling is now well understood for the definition of abstract syntax, formal definition of textual concrete syntax is still a challenge, even

though concrete syntax definition is considered as an important part of meta-modeling [8].

Being able to parse text and transform it into a model, or being able to generate text from a model are concerns that are being paid more and more attention in industry. For instance Microsoft with the DSL Tools [9] or Xactium with XMF Mosaic [10] in the domain-specific language engineering community, are two industrial solutions for language engineering that involve specifications used for the generation of tools such as parsers and editors. A new OMG standard, MOF2Text [11], is also being developed regarding concrete-to-abstract mapping. Although this paper focuses on textual concrete syntaxes, it is worth noticing that there are also research activities about modeling graphical concrete syntax [12] [13].

The aim of the research presented in this paper is to reduce the gap between the fields of grammars and meta-models. We propose a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes whose structure is known in advance to concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations.

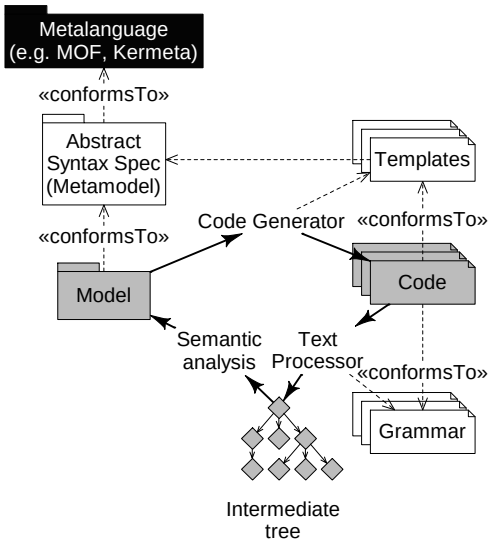


Figure 1: Typical concrete syntax definition and tooling

Figure 1 provides an overview of a typical usage and definition of concrete syntaxes for meta-model-based languages. Abstract syntax is modeled by a meta-model. A language sentence is a model, which is an instance of the concepts defined

in the meta-model, as depicted by the «conformsTo» relationship. Dashed arrows represent dependencies while bold arrows represent data flow. Grayed items represent artifacts that are at the modeling level (e.g. a language sentence - M1 in the MDA terminology [1]), and white items are data necessary to define a language (i.e. the specification for a language - M2 in the MDA terminology). The figure shows a typical solution to manage textual concrete syntax. Usually, this specification is divided into two different parts: producing textual representation from a model, or producing a model from textual representation. To produce textual representation, code generators often instantiate text templates while visiting the model, see for example the AndroMDA tool [14]. Each text template states how a given concept should be rendered in text. To produce models from texts, the usual way is to rely on compiler technologies. Compiler compilers (e.g. the ANTLR tool [15]) create text processors from a grammar. The result is an intermediate tree as a trace of rules triggered to recognize text. Further hand-coded programs need to visit those intermediate trees to translate them into models. One of the drawbacks of that scheme is that both ways (from model-to-text – i.e. synthesis – and from text-to-model – i.e. analysis) have to be kept consistent. As a side effect, when a change occurs in concrete syntax, one needs to update code generation templates, grammar, and semantic analysis accordingly. Moreover, code generation technologies are not well suited for concrete syntax representation. Those technologies emerged for producing code from models, regardless whether target code was actually a representation for another language.

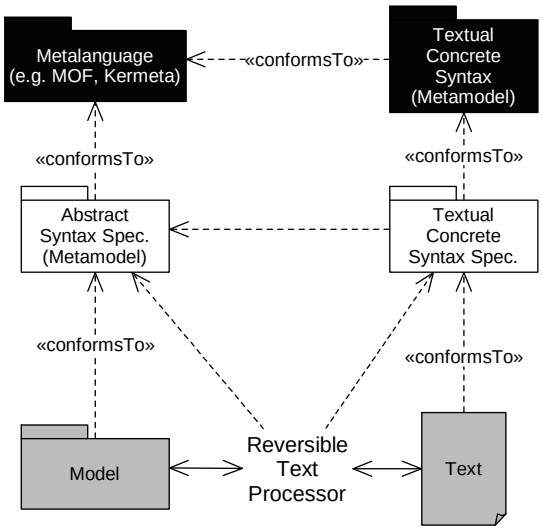


Figure 2: Unifying analysis and synthesis specifications

As shown in Figure 2, we propose in this paper to have a unique specification both for analyzing and synthesizing text, thus reducing the number of maintenance points and simplifying the task of developing semantic analysis. The goal here is limited to

representing a model of a given language into concrete syntax of that same language, so that we do not need as much flexibility as depicted in Figure 1, in which generated code could belong to another language. Indeed, the goal is not to cross the boundaries of languages. Some technologies (such as model transformations) are currently extensively studied (e.g. see [16]), improved and developed regarding transforming a model (sentence) of a given language into a model (sentence) of another language (exomorphic transformations). We have the feeling that it is a better approach to work at the model level rather than at the concrete syntax level to compile languages, since the concern of concrete syntax is not vital in the context of compiling language, and since one could benefit from dedicated technologies as promoted by MDE-like technologies. As an example, one should rather transform his/her UML model into a Java model, which conforms to the Java meta-model, further rendered in text using the Java concrete syntax, instead of directly generating Java code. If the target concrete syntax evolves (as it happens when new keywords appear), code generation has to be maintained while a model transformation that works at the abstract syntax level would still be valid. To specify such a reversible textual concrete syntax specification, we propose a new (DSL) modeling language to put into relation text structures and a meta-models. Driven by specifications written in such a language, automated tools can produce models by analyzing texts, or represent models in textual forms. In this paper, we define such a language by stating its abstract syntax in the form of a meta-model. Of course, such a new meta-model may be used to provide a concrete syntax to itself (see section 5.2).

The material presented in this paper is an evolution of a previous work which was limited to textual concrete syntax synthesis [17]. The ideas presented in this paper were validated by two prototype implementations: TCSSL tools and Sintaks. The TCSSL tools [18] were developed in the software laboratory of the French Atomic Commission (CEA, LIST). They provide a pretty printer for rendering a model as structured text, and they generate a compiler compiler specification for building a model from a text. Sintaks [19] (which stands for syntax in Breton, a Celtic language) has been developed in the context of the Kermeta project [20]. Kermeta is an executable DSL (Domain Specific Language) for metamodeling, which can be used to specify both abstract syntax and operational semantics of a language. Altogether, Kermeta, TCSSL Tools and Sintaks provide a comprehensive platform for model-driven language engineering.

An important point in our approach, and a major difference to the grammarware approach, is that the structure of the abstract syntax (i.e. the meta-model) is known before concrete syntax. This basically means that the abstract syntax does not depend on the structure of the concrete syntax. To the contrary, the description of the concrete syntax depends on the structure of the abstract syntax.

The paper is organized as follows: after this introduction, section 2 examines some related works, section 3 motivates our proposal; section 4 presents our meta-model for concrete syntax, and explains the mechanics which are behind. Section 5 presents two examples which illustrate the way concrete syntax can be modeled and associated to

models of the abstract syntax. Section 6 briefly discusses implementation of the prototypes and finally section 7 draws some general conclusions, and outlines future works.

2 Related works

Many of the concepts behind our work take their roots in the seminal work conducted in the late sixties on grammars and graphs and in the early eighties in the field of generic environment generators. One example is Centaur [21] that, when given the formal specification of a programming language (syntax and semantics), produces a language-specific environment.

The issue of representing information in structured text has been described extensively in the context of compiler construction, in works such as generative grammars [22], attributed grammars [23], Backus-Naur Form (BNF) [24], and Extended BNF (EBNF) [25]. Those results are used by compiler compilers, i.e. programs that take EBNF-like grammars as input, and which generate programs able to analyze texts conforming to grammars. Well known examples of such compiler compilers include Lex/Yacc [26] and ANTLR [15]. A thorough description of compiler construction may be found in [27].

Pratt's pair grammars [28] make two reversible context-free grammars match so that, when one rule describing the source language is recognized, the corresponding rule of the target language is executed. Source and target languages can indifferently be graphs (using a context free graph grammar) or texts. Thus, by matching rules, this approach makes it possible to put in correspondence a graph with a text in a bidirectional way. A model is a kind of graph, so the approach can be applied to make a model match a text (as in [29] where the model is actually a graphical representation). To apply such an approach in the domain of models, one would first need to describe the graph grammar for the considered model. Further on, all non terminal rules should be rewritten to target structured text. However, in our approach, the model structure is already provided as a meta-model. Therefore, we rather propose a single specification that states how a reversible context-free grammar can directly match a meta-model. An important point in our approach is that the structures of the abstract and of the concrete syntax are not necessarily similar.

AntiYacc [30] is an example of generating texts from models, and shares a lot of ideas with our approach. It uses a grammar which extends EBNF and is described by a meta-model as well. The AntiYacc approach may be seen as more flexible than ours (e.g. it features powerful expressions for model querying) but it addresses only half of the problem. Despite its name, there is no seamless way to transform the abstract-to-concrete specification into a concrete-to-abstract specification that could be used by a tool like Yacc.

XMI (XML Metadata Interchange) [6] provides an example of a reversible mapping between models and texts. The primary goal of XMI is to offer a standard way of serializing models, mainly for model interchange between tools. Hence, the concrete syntax of XMI is hard to read and cannot be used on a large scale by humans for creating models. A major strength of XMI is its compatibility with XML tools (most notably XML parsers). HUTN (Human Usable Textual Notation) [31] is a standard for deriving human-friendly generic concrete syntaxes from meta-models. HUTN syntax is much easier to read than XMI; however it cannot be fully customized. Our work is close to HUTN, but in a more general context, as we support user-defined concrete syntax. In the end, HUTN is a good use-case for Sintaks.

There is currently a lot of interest in the modelware community about establishing bridges between so-called technological spaces [32]. For instance M. Wimmer and G. Krammler have presented a bridge from grammarware to modelware [33], whereas M. Alanen and I. Porres discuss the opposite bridge from modelware to grammarware, in the context of mapping MOF meta-models to context-free Grammars [34]. A. Kunert goes one step further and generates a model parser once the annotated grammar has been converted to a meta-model [35]. The Textual Editing Framework (TEF) [36] is a recent approach that makes possible to define textual concrete syntaxes. The tool can translate the specification into a parser that includes semantic analysis. The generated parser can be used in background by the Eclipse IDE, which is automatically customized by the framework for the described language. TCS (Textual Concrete Syntax) [37] is another example of bi-directional bridge, based on a DSL for specification of concrete syntax.

While a grammar could be considered as a meta-model, the inverse is not necessarily true, and an arbitrary meta-model cannot be transformed into a grammar [38]. Even meta-models dedicated to the modeling of a given concrete syntax (such as HUTN presented earlier) may require non-trivial transformations to target existing grammarware tools. We discuss some of these issues in previous work, where we have experimented how to target existing compiler compilers to generate a parser for the HUTN language [39]. A similar experience, turning an OMG specification (OCL) into a grammar acceptable by a parser generator [40] has been described by D. Akehurst and O. Patrascoiu.

As we have seen, the issue of transforming models into texts, and texts into models has been addressed mostly as two different topics. In this paper, we explore a bidirectional mapping by defining a meta-model for the specification of textual concrete syntax in a context where abstract syntax is also represented by meta-models. The transformations described in this paper (from model-to-code, and from code-to-model) are symmetric, and their effect can be reversed by each other. In the context of this paper, we call analysis the process of transforming texts into models, and synthesis the process of transforming models into texts.

We are experimenting with a new way of building tools for programming languages (such as compilers or IDEs) by using meta-models which embed results

from the language theory directly in the modelware space. The novel contribution of this research is combining the specification of the two approaches (text-to-model and model-to-text) in a single approach, i.e., inconsistencies between text-to-model transformations and model-to-text transformations are avoided that way.

3 Modeling the mapping between abstract syntax and concrete syntax

Before detailing the mapping, it is worth reminding general principles of text parsing.

3.1 General principles of text parsing

A text parsing process starts with lexical analysis. The input stream is split into a sequence of tokens (number, keyword, plain text, etc), usually described by regular expressions. A token is associated to a type (number, comment, keyword, etc.), a text (e.g. "12.85" for a number), and eventually a location (e.g. a line and a colon). A tool fulfilling this function is called a lexical analyzer, in short a lexer.

The process goes on with syntactic analysis. The token stream obtained from the lexical analysis is analyzed according to a grammar of the language under study, usually defined in the form of EBNF-like rules. Actions may be embedded in the grammar rules (attributed grammar), so that one may build an abstract syntax tree (AST) that may be later walked through to generate code. A tool fulfilling syntactic analysis is called a parser (for simplicity, we do not detail scanner-less parsing here).

There are two main families of parsers: top-down parsers and bottom-up parsers.

- Top-down parsers (LL parsers - from Left to right scanning using Leftmost derivation) attempt to trigger the topmost rule (the main rule), and move down into grammar rules. For instance, if *expression* is the topmost rule, a LL parser will trigger the *expression* rule and then choose among the alternatives such as *addition* or *subtraction*. *Left recursivity*, i.e. a rule potentially calling itself as a first sub-rule, must be avoided.

- Bottom-up parsers (LR parsers - from Left to right scanning using Rightmost derivation in reverse), also known as shift-reduce parsers, will store intermediate tokens into a stack up to a rule is matched. For instance, it is only when the token stack contains a number, a '+' sign, followed by another number, that the *addition* rule will be recognized as an *expression* rule. However, these parsers are difficult to produce by hand.

Parsers usually drive their associated lexer; that is, they consume tokens while they analyze rules. An important issue is that rules may be conflicting, i.e. different rules

(and alternatives) may be applied to recognize the same text. For instance, at the moment a parser consumes a number terminal, it cannot decide whether the triggered rule should be *addition* or *subtraction*. To solve that problem, two options are possible: look-ahead and backtracking.

Look-ahead parsers can analyze more than one token at a given moment. While analyzing an *expression*, such parsers can choose between *addition* and *subtraction* by looking at the next token, which should be either a '+' or a '-' terminal. Depending on the grammar complexity, the number k of tokens to be accessible in advance is different, but the more important that number is, the more complex and the less efficient the parser is. The parser is said to be LL(k) or LR(k). Nowadays, one can easily find LL(1), LL(*), and LR(1) parser generators.

Backtracking parsers, such as the TXL parser [41], follow a try-and-error scheme. Whenever a rule fails (for instance because of an un-matching token), such parsers abandon the current rule, backtrack, select another rule, and try again to parse the input text. For instance, a backtracking parser will choose first the *addition* rule each time it encounters a number terminal, "hoping" that the right rule was chosen. If the next token is a '-' terminal, then *addition* rule fails, and the parser backtracks to the point of the rule choice, and starts again analysis using the *subtraction* rule alternative. Unfortunately such a process is rather inefficient compared to the previous one.

Our goal has not been to reinvent a text format specification or parsing technique, but to extend the above-described techniques so that they can specify both analysis and synthesis. Regarding analysis, our extension targeted the creation of a model while parsing a text representation, i.e. semantic analysis. Regarding synthesis, extension was about making the process reversible. We chose to compare both look-ahead and backtracking approaches in our prototype implementations. Regarding text analysis, the TCSSL tools generate an ANTLR compiler compiler specification from a meta-model and a textual concrete syntax specification. Since ANTLR is an LL(k) parser generator, the generated text analyzer follows LL(k) rules: parsing is efficient but grammars must neither be left recursive, nor ambiguous for a k lookahead. This last point requires the ambiguities as detected by the ANTLR tool to be transformed into errors that are understandable from the perspective of the genuine textual concrete syntax specification. Note that ANTLR can compute and use the minimal value for k locally to a rule. To the contrary, Sintaks interprets the textual concrete syntax specification while analyzing an input text by executing a backtracking mechanism. The scope of possible grammars is thus larger since ambiguous grammars are acceptable: ambiguities (among choices) are resolved depending on the order those choices were declared. However, for sake of ease of implementation, Sintaks is based on a top-down recursive descent: as for TCSSL tools, left recursivity still needs to be avoided. As said above, the major drawback is that analysis is slower (around 6 times, highly depending on the grammars), which is not really an issue as long as the goal is building languages by prototyping. Nevertheless, this approach

could scale when it comes to create and test a textual concrete syntax, following the philosophy of the TXL engine [41].

3.2 Abstract syntax versus concrete syntax

Defining a language can be decomposed into three related activities: defining the syntax, the semantic domain, and the mapping between syntax and semantic domain. D. Harel and B. Rumpe give a good introduction to the issues surrounding these activities in their paper about defining complex modeling languages [42]. In this paper we focus on syntax definition; defining semantic domain and mapping syntax to semantic domain is out of the scope of the work presented here.

Syntax can be further decomposed into abstract syntax and concrete syntax since "surface structure does not necessarily provide an accurate indication of the structure and relations that determine the meaning of a sentence" ([43] on page 93). Abstract syntax describes the concepts of a given language independently of the source representation (concrete syntax) of that language and is primarily used by tools such as compilers for internal representation. Concrete syntax, also called surface syntax, provides a user friendly way of writing programs; it is the kind of syntax programmers are familiar with.

Object-oriented meta-modeling languages (such as EMOF, Ecore or Kermeta) can be used for representing abstract syntax; concepts of languages are then represented in terms of classes and relations. A given program can be represented by a model conforming to the meta-model which represents the abstract syntax of the language used to write the program. Writing this program, in other words building the model which represents the program, requires some way to instantiate the concepts defined in the meta-model.

This can be achieved either at the abstract syntax level, or at the concrete syntax level. The difference is that in the first case, the user has to manipulate the concepts available in the meta-modeling environment (for instance via reflexive editors) while in the second case, the user may use a surface language which is made of those concepts. While the end-result is the same, it is much simpler for users (such as programmers) to write programs in terms of concrete syntax, rather than using instances of meta-modeling concepts.

For textual languages, there must be some way to link the information in the text (the concrete syntax) with the information in the model (the abstract syntax). We have seen in the previous subsection that the issue of analyzing text to produce abstract syntax trees had already received much attention in the compiler community. Most notably, efficient tools are available to generate parsers from EBNF-like grammars. Unfortunately, these generated parsers produce abstract syntax trees, leaving a gap with the abstract syntax defined by meta-models (graphs related), and further ad-hoc hand-coding is required in order to perform semantic analysis (see Figure 1).

An example of such a time consuming task is type checking, which actually matches name-based references to other elements that are defined elsewhere. Typical examples are typed programming languages, where one may declare variables of a certain type, which can be considered correct if and only if that type is declared elsewhere in the code. Other examples are method and operator calls. It is again the task of type checking to actually find the called operation according to its name and the inferred type of the arguments. Again, type inference is a concern of semantic analysis.

In practice, structure of concrete syntax often influences abstract syntax. Some even derive the abstract syntax from the concrete syntax, following a bottom-up approach like in [34]. However, for the same language, one may want to define different concrete syntaxes (e.g. UML models that can take a graphical appearance, but also textual XMI or HUTN shapes). If different surface syntaxes are provided, a bottom-up approach to abstract syntax specification will lead to different abstract syntaxes for the same language. In our approach, we propose to improve text structure definition to support manipulation of predefined abstract syntaxes given under the form of meta-models.

3.3 Code generation versus textual concrete syntax

Code generators are usually built for one specific source language (e.g. UML), and for one specific target language (e.g. Java). This two-dimension dependency outnumbers the necessary code generators by a Cartesian product factor. Moreover, this architecture can raise several problems when a source model needs to be customized, for instance when using a profile.

We believe that a better approach would be to pass through an intermediate model (i.e. an abstract syntax instance). In the example of a Java code generation from an UML model, the approach we propose would be the following: the UML model (instance of the UML metamodel [44]) would be transformed into a Java model (instance of a Java metamodel such as [45]) through a model transformation, and then the Java model would be synthesized into Java text files by mean of a text synthesizer. A striking advantage of this approach is that the semantic domain translation is achieved by a model transformation, which is a dedicated technology, while text synthesizers would need to deal with the concrete syntax of the target language. The process distinguishes two different tasks (transformation and synthesis) that are performed using appropriate approaches.

In case of compilers, a source textual specification expressed in a language A has to be transformed into a target textual specification in language B. To perform this task using a model-based approach so as to benefit from model transformation technology, an additional step is required: source code must be analyzed into a model, for instance by mean of a text processor. A problem with that approach is that, for a given language L, one needs to provide either a text analyzer, in case L is the source language; or a text synthesizer, in case L is the target language; or even both, should L

play the role of both source and target languages. In this latter case, which appears for instance in round-trip engineering processes, analyzer and synthesizer have to be consistent. This motivates the unified specification proposed in this paper.

The process for compiling a specification given in language A into a B specification is given in Figure 3. In the MDA terminology, white represented models at the M2 level (meta-level) and grayed models reside at the M1 level. Examples for A/B couples are Java/C++, UML (e.g. represented with HUTN or XMI)/Java, and B/C++. One may notice that in these examples Java may play both the role of a source or a target language. A textual A specification is analyzed into an A model. The A model is transformed into a B model, and the B model is thus synthesized into a B representation. It is interesting to see that if the A/B transformation is reversible, one may reverse the complete process (from B textual representation to A textual representation) with no additional development.

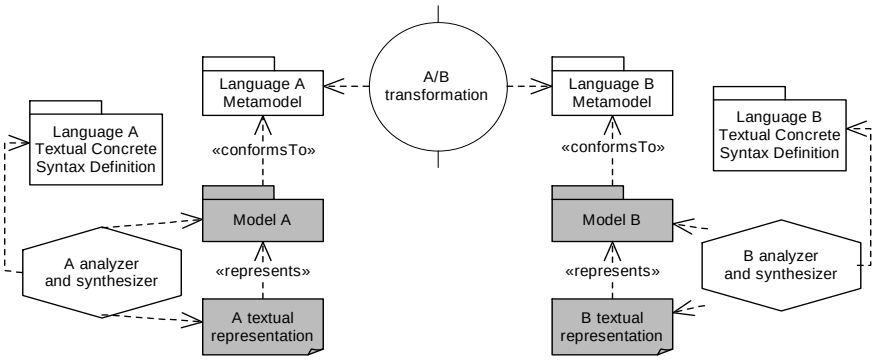


Figure 3: Approach to Code Generation based on Textual Concrete Syntax

In the next sections we propose a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes to concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations.

3.4 Requirements for mapping abstract syntax to concrete syntax (and the reverse)

Modeling the mapping between abstract and concrete syntax means expressing how a given piece of information can either be stored in an object model (considering that we used object-oriented meta-languages to define abstract syntax) or represented in text (as we focus on textual concrete syntax). One must consider that there is no one-to-one mapping between abstract and concrete syntax, and further, that there is no single solution either to store information in a model or to represent it in text.

The multiple ways to capture information (e.g. the structure of abstract syntax) in an object model are genuinely addressed by object-oriented meta-models. Elementary information can be modeled as a class, an attribute, a relation, or a role. Attributes, relations, and roles may be shared among classes (and relations) in presence of inheritance. Storing more complex information in a given model is then achieved by creating clusters of instances of the modeling-elements defined in their corresponding meta-model.

As we have seen earlier, representing information in text follows structures known as grammars. Building a mapping between models and texts therefore implies understanding how information can be mapped between models (graphs of instances) and texts (sequences of characters).

An instance of modeling-element, following the example of an object, is characterized by an identifier and a state. In an object model, state is stored in the slots of the instances (i. e. values, either of primitive types, or references to other instances of modeling-elements).

Going from abstract syntax to concrete syntax (or the reverse) is then a matter of explaining how pieces of abstract syntax (i.e. values held by slots of instances of modeling-elements) are to be serialized (or conversely de-serialized) to pieces of concrete syntax (actually character strings, as we target textual concrete syntaxes).

This means that in addition to what traditional text analysis tools provide (e.g. terminal, sequences, and alternatives), concrete syntax definition should also offer the possibility to state how individual slots of instances of modeling-elements are mapped to concrete syntax.

3.5 Toward an implementation for the mapping

Let's consider the following meta-model of a language which defines models as a collection of types where types have attributes, which in turn have a type.

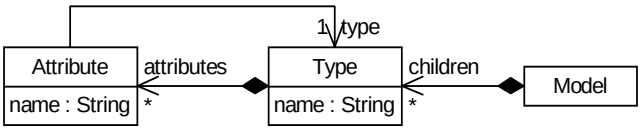
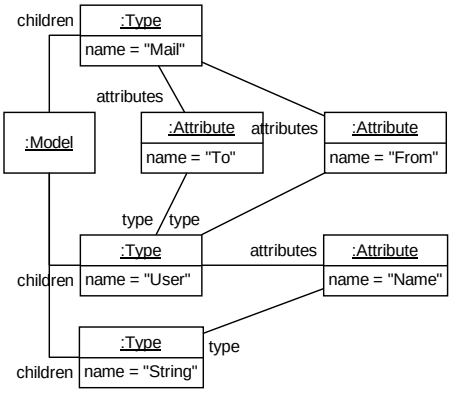


Figure 4 : Meta-model of abstract syntax for a simple language

A model instance and a typical concrete syntax may be:




```
Type Mail {
   From : User
   To : User
}

Type String;

Type User {
   Name : String
}
```

Figure 5: The same model represented as an instance diagram and using a specific textual concrete syntax

The concrete syntax for the meta-model shown in Figure 4 can be described in plain English as shown in Figure 6:

> The text represents a `Model` instance. The list of the `Model`'s `children` is represented. A `Type` is declared by a 'Type' keyword followed by the `name` of the `Type` and the collection of its `attributes`. If not empty, the collection of `attributes` is enclosed by curly braces; an empty collection is specified by a semi
> colon. An `Attribute` is represented by its `name`, followed by a colon and the `name` of its `type`. Notice that the notation allows forward and backward references (e.g. to `User` in the `Mail` type declaration and to `String` in the `User` type declaration). Representations for attributes *should* be placed on a dedicated line.

Figure 6: Textual Concrete Syntax in Plain English

The meta-model on Figure 4 captures the abstract syntax (the concepts of the language, and their relations). The goal of our approach is to describe a concrete syntax following the example of that one given in plain English in Figure 6, but in a formal way, once given a meta-model that structures the abstract syntax. In our approach, we have introduced the concept of *template* which is associated to meta-classes. Each instantiable modeling-element (i.e. each concrete meta-class) defines a *template* whose role is to organize the serialization (de-serialization) of the instances

of its (meta-)features, regardless if they are defined in the meta-class or inherited. Each (meta-)feature further defines a *feature-mapper*, which knows how to fill a given slot with elementary data coming from a text (and conversely, how to generate text from the data).

Templates should also embed facilities for iteration (repeating the same sequence a given number of times) and for alternatives (alternate textual representations according to a given constraint). Moreover, it must be possible to define the value of a slot depending on an alternative; for instance, setting a slot to true or false depending on the occurrence of a given sequence of tokens in the textual representation (or conversely, generating a given sequence of tokens depending on the value of a slot). Alternatives can also be used to specialize children class descriptions based on a single mother class.

Beyond the basic alternative and sequence capabilities, the various mapping options are described by 3 major properties of the feature-mappers: the kind of values held by the slots, the multiplicity of the information to be represented, and the way to share and parameterize feature-mappers. These properties are presented in detail in the following subsections.

3.5.1 Kinds of data

Slots may contain three different kinds of data:

- *Attributes values* which refer to primitive types; i.e. either data types (such as String, Integer, Real, and Boolean) or enumerations. A single-value feature-mapper, with automatic type translations (to and from text) can handle these attributes.

- *Compositions* which physically embed the slots of the contained instance of modeling-element into a slot of the containing instance. Representing such a relation can be realized straightforwardly by embedding the template of the owned meta-class into the template of the owning meta-class.

- *Simple references* are a little bit more subtle a problem, and denote that a source instance of a modeling-element refers to a target instance of another modeling-element, using a given key which is in fact a specific slot value. In practice, the textual representation of the referenced instance of modeling-element may appear after the representation of the reference, as in the example presented in Figure 5 where `Type String` is represented *after* the `Attribute Name` of `Type User`, whose type is `String`. This referencing capability is also required to implement bidirectional associations (A references B which in turn references A).

3.5.2 Multiplicity of the data

Features are either mandatory or optional, and single or multiple. In the meta-model stating the abstract syntax of the considered language, the optional nature of a feature is rendered by the lower bound of the multiplicity (0 for optional, and 1 for mandatory), and the single/multiple nature is rendered by the upper bound (1 or *).

- Representing the mandatory/optional nature can be done by reusing the alternative rule: optional information will be represented by an alternative with an empty branch.

- Representing the single/multiple nature can be done by using either a single-value feature-mapper or an iteration.

3.5.3 Shared and parameterized feature-mappers

In practice, properties for different classes often share the same concrete syntax. This is especially true regarding inherited properties. Thus, it may be interesting to introduce *shared* definitions in the feature-mapper to address this reusability concern, following example of EBNF non-terminal rules that can be called from different rules. Feature-mappers may then be defined outside the scope of given meta-class template, and further called by several meta-class templates; in the same way a procedure may invoke a sub-procedure in an imperative language.

Two cases may be distinguished, representing whether a feature-mapper knows the feature to map or not. The first case typically allows reusing the mapper for the same feature within an inheritance hierarchy of meta-classes. The second case (we talk about *parameterized* feature-mappers) permits to share the same feature-mapper not only by different templates but also by different features, even across meta-class hierarchies.

4 Modeling concrete syntax

As seen in the previous section, when defining a language, the meta-model of the abstract syntax has to be complemented with concrete syntax information. In our case, this information will be defined in terms of a model conforming to a meta-model for concrete syntax, which has to be used as a companion of the meta-model already used for defining the abstract syntax of the language under specification.

Fully defining the syntax of a language is achieved by combining the abstract syntax as supplied by a meta-model with one or more concrete syntax specification as supplied by a model conforming to the meta-model introduced in next section. The effect of parsing text (conforming to a concrete syntax model) is to create a model (conforming to the abstract syntax meta-model). Conversely, the text can be synthesized from the model.

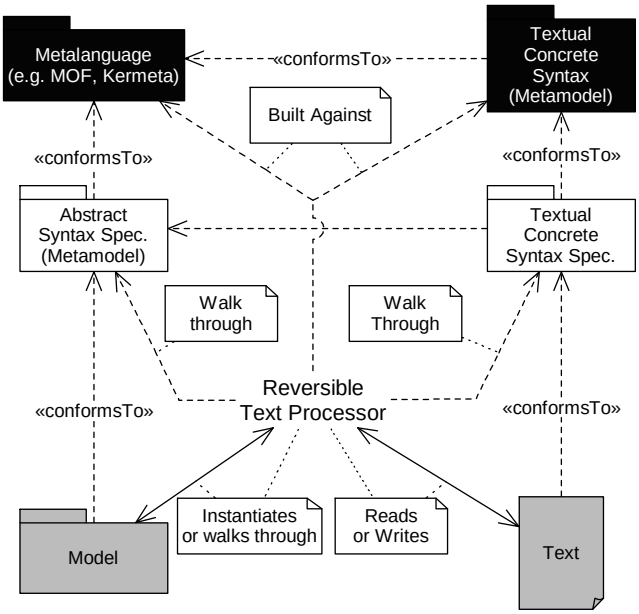


Figure 7: A model-driven generic machine performs the bidirectional transformation

As shown in Figure **7**, the reversible text processor is configured by the meta-model of the language and a textual concrete syntax specification. Those two models are supplied as instances of a meta-meta modeling language (e.g. MOF or Kermeta) and of a textual concrete syntax meta-model. The goal of analysis is to construct a model by instantiating the meta-model while reading a text specification. The goal of synthesis is to generate a text while walking through a model.

During synthesis, text is generated by a generic synthesizer which operates like a model-driven template engine. The synthesizer visits the model (conform to the abstract syntax meta-model) and uses the presentation information available in the concrete syntax model (conform to the concrete syntax meta-model) to feed text to the output stream.

Interestingly, both processes of analysis and synthesis are highly symmetric, and since they share the same description, they are reversible. Indeed, a good validation exercise for the prototype is to perform two synthesis-parse sequences, and observe that there are no significant differences in both generated texts.

4.1 Overview of our proposal

Our meta-model for concrete syntax is displayed on Figure 8. Given concrete syntax has a top-level entry point, materialized by the root class which owns top-level

rule fragments and meta-classes. A model of concrete syntax is built as a set of rules (the sub-classes of abstract class *Rule*). The bridge between the meta-model of a language and the model of its concrete syntax is based on two meta-classes: *Class* and *Feature* respectively referencing the class of the abstract syntax meta-model and their properties. Class *Template* makes the connection between a class of the meta-model and its corresponding rules. Class *Value* (and its sub-classes) and class *Iteration* make the connection between the properties of a class and their values. Class *Iteration* is used for properties whose multiplicity is greater than 1. The remaining classes of the meta-model provide the usual constructions for the specification of concrete syntax such as terminals, sequences and alternatives.

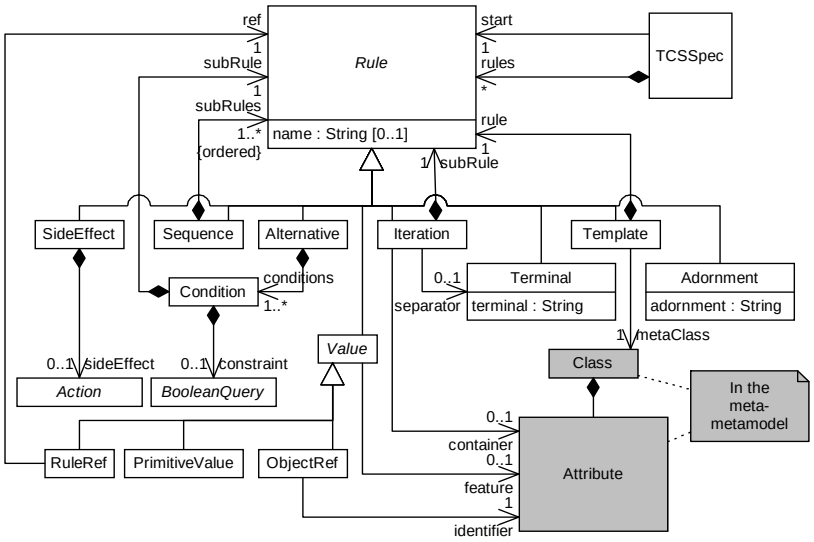


Figure 8: Overview of the meta-model for concrete syntax

The following sub-sections detail the semantics (in plain English) associated with each element of our concrete syntax meta-model, from both analysis and synthesis prospectives. We illustrate each concept using the example of section 3.5 by showing corresponding excerpts of Figure 6.

4.2 Template rule

A Template rule makes the connection between a class of the meta-model (property *metaClass*) and a sub-rule.

Analysis semantics: The template specifies that an object should be created. The meta-class is instantiated and the object is set as the current object. The sub-rule is

invoked and the current object is initialized. If an error occurs the current object is dismissed.

Synthesis semantics: The template specifies which object to serialize. The sub-rule is invoked to generate the corresponding text.

Example: An `Attribute` (*metaclass*) is represented by (*sub-rule follows*) its `name`, followed by a colon and the `name` of its `type`.

4.3 Terminal rule

A terminal rule represents a text whose value is constant and known at modeling time. The text value is stored in the property *terminal* of type *String* in class *Terminal*.

Analysis semantics: The text in the input stream must be equal to the terminal value. The text is simply consumed. If the text does not correspond an exception is thrown.

Synthesis semantics: The terminal value is appended to the output stream.

Example: …a 'Type' keyword… *or* …a colon…

4.4 Sequence rule

A sequence rule specifies an ordered collection of sub-rules. A sequence has at least one sub-rule.

Analysis semantics: The sub-rules are invoked successively. If any sub-rule fails the whole sequence is dismissed.

Synthesis semantics: The sub-rules are invoked successively.

Example: A `Type` is declared by a 'Type' keyword (*sub-rule 1*) followed by the `name` of the `Type` (*sub-rule 2*) and the collection of its `attributes` (*sub-rule 3*).

4.5 Iteration rule

Iterations specify the repetition of a sub-rule an arbitrary number of times. An iteration uses a collection (property *container* of type *Feature*), and may have a terminal to be used as a separator between elements (property *separator* of type *Terminal*). However, for complex iteration expression, one may refer to derived properties if model repository supports it.

Analysis semantics: The sub-rule (and separator, if specified) is invoked repetitively, until the sub-rule fails. For each successful invocation the collection specified by the container feature is updated.

Synthesis semantics: The sub-rule is applied to each object in the referenced collection, and the optional separator (if specified) is inserted between the texts which are synthesized for two consecutive elements.

Example: The list of the `Model`'s `children` (*container*) is represented. A `Type` (*i.e. the meta-class that isthe type of the* `Model::children` *property*) is declared by… (*the sub-rule*)

4.6 Alternative rule

Alternatives capture variations in the concrete syntax. An alternative has an ordered set of conditions which refer each to a given sub-rule. A boolean expression in a query language may constrain choice for the correct condition. Queries may be written using languages like OCL, Kermeta, MTL, Xion, or even JMI Java code; for instance TCSSL tools uses EMF Java code, and Syntax defines its own language for comparing a feature with a value, or testing the object's metaclass.

Analysis semantics: This is the most complex operation. Often there is no clue in the input stream to determine the condition (in the sense defined in the meta-model for concrete syntax) which held when the text was created. It is therefore necessary to infer this condition while parsing the input stream. The simplest (but also the most time consuming) solution, applied by Sintaks, is to try each branch of the alternative until there is a match. It is worth noticing that the ordered collection of conditions can also be used to handle priorities between conflicting sub-rules. Another solution, as implemented by TCSSL tools and by most text analyzers, is to choose alternatives depending on the next lexemes (see section 3.1). Conditions' queries should be enforced once the analysis is completed. If Sintaks does not make use of the conditions' queries at analysis, TCSSL tools force them to be a comparison, with an attribute access at its left-hand side. At analysis, the comparison is interpreted as an affectation: the slot corresponding to the attribute value is set to the result of the computation of the right-hand side of the condition. More experiments should decide whether constraint solving or constraint enforcement are helpful in choosing an alternative.

Synthesis semantics: The conditions are evaluated in the order defined in the collection, and the first one which evaluates to true, triggers the associated rule.

Example: (*Condition 1:*) If not empty (*constraint for condition 1*), the collection of `attributes` is enclosed by curly braces (*sub-rule for condition 1*); (*Condition 2 :*) an empty (*constraint for condition 2*) collection is specified by a semi-colon (*sub-rule for condition 2*).

4.7 Primitive value rule

The rule *PrimitiveValue* specifies that the value of a feature is a literal. The type of the referenced feature should be a primitive type such as Boolean, Integer or String.

Analysis semantics: The literal value corresponding to the type of the feature is parsed in the input stream. The result is assigned to the corresponding feature of the current object unless the type conversion failed.

Synthesis semantics: The value of the feature in the current object is converted to a string and appended to the output stream.

Example: the `name` of the `Type` (*feature*)

4.8 Object reference rule

This rule implements the de-referentiation of textual identifiers to objects. Identifiers (such as names or numbers) are used in texts to reference objects which can bear an attribute whose value contains such identifiers.

Analysis semantics: The reference which is extracted from the input stream is used as a key to query the model so as to find a matching element. If there is a match, the parser updates the element under construction. If there is no match, the parser assumes that the referenced item does not yet exist (because it might be defined later in the text), the referencing is deferred in a "to-do" list, and will be tried again next time the model is changed (instantiation or slot update). By the end of the parsing process, the "to-do" list has to be empty unless there is a parsing error.

Synthesis semantics: The identifier is printed to the output stream.

Example: An `Attribute` is represented by (…) the `name` (*identifier*) of its `type` (*feature*). Notice that the notation allows forward and backward references (e.g. to *User* in the *Mail* type declaration and to *String* in the *User* type declaration).

4.9 Rule reference rule

The rule *RuleReference* references a top-level template, stored under the root of the concrete syntax model.

Analysis semantics: The *ref* rule is triggered and the result is assigned to the feature of the current object.

Synthesis semantics: The *ref* rule is triggered.

Example: We will detail examples in the following section.

4.10 Side effect rule

An action is an instruction that has a certain impact on the model. Action may be written in any language capable to impact a model. Examples of such languages are Kermeta, Xion, and JMI Java code. OCL cannot be used as such since it is a side effect free language.

Analysis semantics: The action is performed on model. The contextual object is that one of the rule.

Synthesis semantics: Action is merely ignored.

Example: We will detail an example in section 5.2.

4.11 Adornment rule

In order to have proper representations for models, one needs to specify non-significant characters such as space or carriage return. This allows to render models not on a single line.

Analysis semantics: Adornment is merely ignored.

Synthesis semantics: The adornment is added to the rendered text. To limit the number of necessary adornments, a white space can automatically be introduced after each terminal, object reference, and primitive value rule application. As shorthand, a \t adornment indents text while a \b adornment unindents it.

Example: Representations for attributes *should* be placed on a dedicated line.

4.12 Towards more complex mappings

The proposed solution does not claim to solve all the problems encountered in compiler construction. In particular, it does not avoid a possible necessity of multiple pass, even if the "to-do" list actually helps. For instance, this is necessary in order to perform full type checking. This part presents a solution to integrate multiple pass analysis in the proposed solution.

The main idea is to keep the classical way of solving the multiple-pass problem, which is transforming decorated abstract syntax trees. The advantage of our approach is that abstract syntax trees are models, conforming to metamodels. This offers the possibility to formally define passes and available decorations: decorations are no longer typeless key-value pairs, but attributes that are added to the metamodel. Thus, passes are merely model transformations exploiting information from decoration attributes. Once processed, this information should still appear in the model to which those decorations are removed.

To formally add attributes to a metamodel (i.e. a decoration), one might use package merge mechanism [1]: this technique uses refinement of modeling elements, to add compatible additional features, such as adding attributes to a class. It works like inheritance, but at the metamodel level: in case a model `m` conforms to a metamodel `cm` that merges metamodel `am`, `m` also conforms to `am`. So, to add a new decoration attribute to a class, one should make a new metamodel merge the genuine metamodel, and refine classes by adding the new decoration attributes.

An n-pass architecture includes then n−1 refinements of the main metamodel, a TCSSL specification for the $n-1^{th}$ refinement, and n−1 model transformations. Note that for the approach to be valid both at analysis and synthesis time, the model transformations have to be bidirectional. As an example, Figure 9 shows a 2-pass architecture. The genuine metamodel is refined to include specific decorations that a dedicated model transformation handle to construct the final model. The TCSSL specification actually describes the concrete syntax for the decorated metamodel.

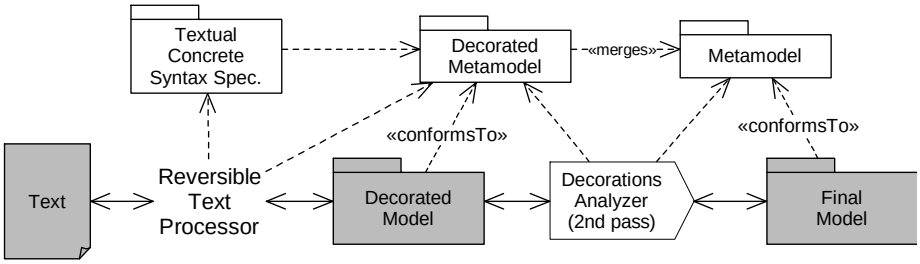


Figure 9: A 2-pass architecture

The following section shows how the concrete syntax meta-model is used for specifying concrete syntax.

5 Examples

The following examples illustrate our approach. The first example is the formal description of the example given in section 3.5. The second example gives an overview of the specification of a concrete syntax to the meta-model of Figure 8.

5.1 A very simple example of concrete syntax specification

We use our meta-model of concrete syntax to specify a textual representation of the language presented in Figure 4. In the example of concrete syntax given earlier (see Figure 5), there is no specific materialization of the *model* meta-class in the text. We propose in Figure 10 the textual concrete syntax for that language, and we detail the analysis and the synthesis processes.

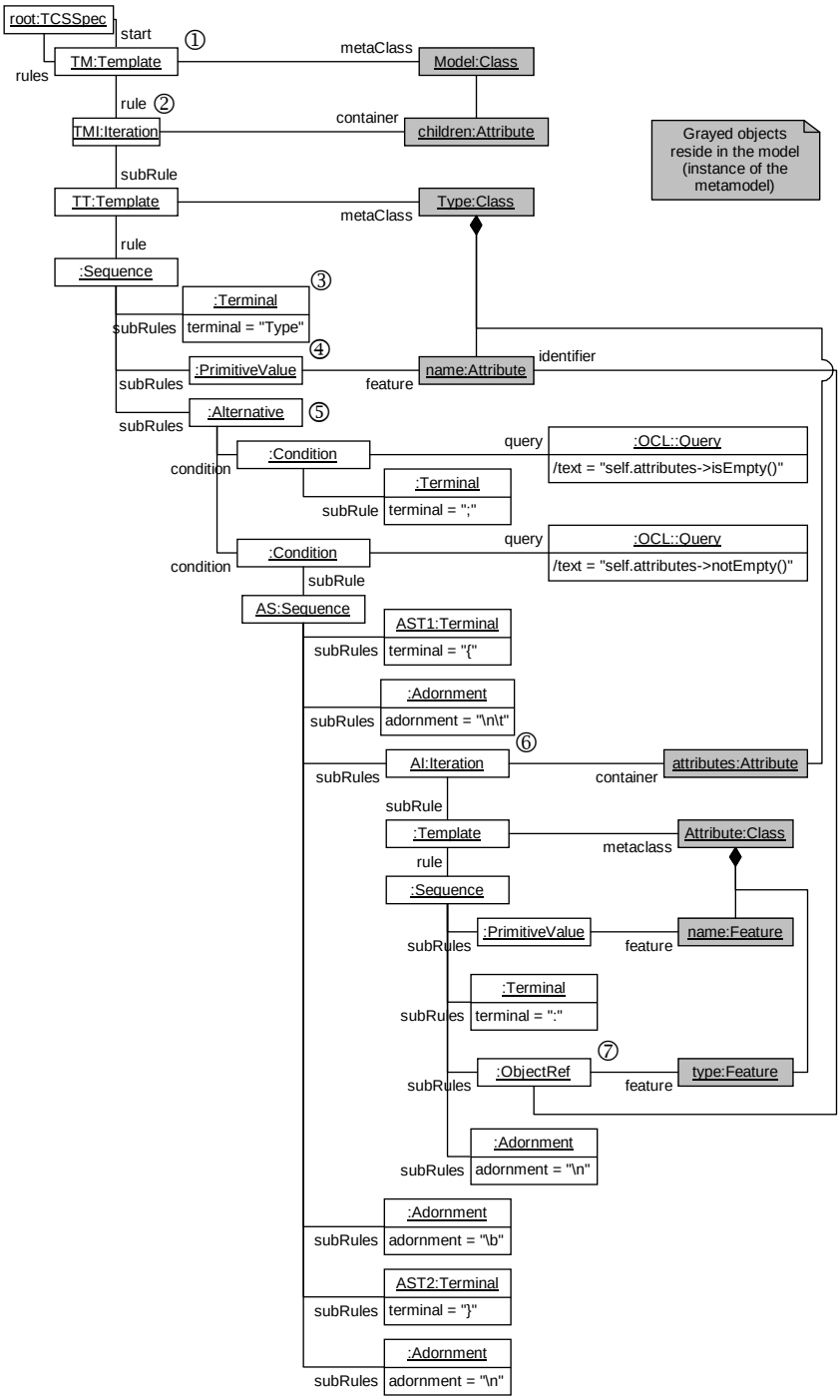


Figure 10: Textual concrete syntax specification as instances of

Figure 8

In this model, there is only one top-level rule which describes the concrete syntax of the language. The model ① is built as a cascade of rules. The model starts with a `TM` iteration over types ②. The sequence explains that types start with the keyword "Type" ③, followed by a name ④, and then an alternative ⑤ because types may have a collection of attributes. The collection of attributes is expressed by an iteration ⑥, which in turn contains a sequence made of a name, followed by a separator (terminal ':') and finally a reference to a type ⑦. Attributes, when present, are delimited by curly braces. As explained in section 3.5, and according to the meta-model of Figure 4, ④ is a mandatory attribute-value feature-mapper (as `Type.name` is an attribute with a `1..1` multiplicity), ② is an optional multiple composition feature-mapper (as `Model.children` is a composition with a `0..*` multiplicity), ⑥ (in association with ⑤) is an optional multiple composition feature-mapper (as `Type.attributes` has a `0..*` multiplicity), and ⑦ is a mandatory reference feature-mapper (as `Attibute.type` is a reference with a `1..1` multiplicity).

Often, it is desirable to share some part of the concrete syntax. Therefore templates do not have to be nested, and can be defined individually at the top level of the model of the concrete syntax. The following picture represents such a variation, for the same concrete syntax. Links between independently defined templates are realized with rule references (*RuleRef*). Actually, ② and ⑥ are now defined independently as shared feature-mappers (as introduced in section 3.5.3).

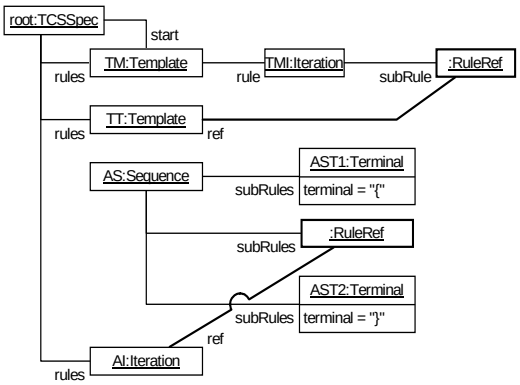


Figure 11: Variation with top-level reusable template

Both representations are totally equivalent. The parsed models or the generated texts are identical.

If one provides a textual concrete syntax for the meta-model given in Figure 8, the specification given above may be written in textual form rather than a raw meta-model instance as depicted by object diagrams as shown in Figure 10 and in Figure 11. The specification mentioned above is represented in Figure 12 using a typical concrete syntax for our textual concrete syntax specification language. That particular

syntax is introduced in next section. One may notice that such a textual specification for textual concrete syntax is much more readable than the object diagrams represented above as it is specialized (one could even talk of a domain specific concrete syntax), compared to general purpose representation formalisms such as object diagrams or XMI.

```
start template (TM) handles Model do
      iterate (TMI) handles Model::Children do
call TT
template (TT) handles Type do
      < "Type" >
      value handles Type::name
      choice
            when self.attributes->isEmpty() do < ";" >
            when self.attributes->notEmpty() do call AS
      end
begin (AS)
      < (AST1) "{" >
      <@ "\n\t">
      call AI
      <@ "\b">
      < (AST2) "}" >
      <@ "\n">
end
iterate (AI) handles Type::attributes do
      template handles Attribute do begin
            value handles Attribute::name
            < ":" >
            reference handles Attribute::type
                  with Type::name
            <@ "\n">
      end
```

Figure 12: Concrete syntax specification in textual form

In order to better understand the text synthesis process, we describe here the behavior of the generic analyzer and synthesizer in synthesis mode. The language description is provided by the meta-model shown in Figure 4 and by the textual concrete syntax specification as introduced above. The model to serialize is the model represented by the object diagram of Figure 5.

The main object must be provided. In this particular case, it is the object instance of the `Model` metaclass. The `TM` start rule is then invoked to produce the text, taking the `Model` instance as the contextual object. The `TM` invokes the `TMI` iteration rule which calls the `TT` sub-rule for all the `children` of the `Model` instance (i.e. the `Mail`, the `User` and the `String` `Type` instances). For each one of these iterations,

the contextual object is the `Mail`, the `User` and the `String` instance, respectively. The invocation of the `TT` rule over the Mail contextual instance first writes '`Type`' to the output stream, then the `name` of the instance (i.e. '`Mail`'), with a whitespace automatically inserted in-between. Since the `Mail` instance references some `attributes` (as detected by the `self.attributes->notEmpty()` OCL expression – `self` references the contextual instance), the `AS` sub-rule is called, again with the `Mail` instance as the contextual instance. The call appends a "`{`" to the output stream (according to the `AST1` rule), a carriage return and an indentation (according to adornment), and then invokes the `AI` rule. The `AI` rule iterates over the `attributes` of the `Mail` instance, namely the `From` and `To Attribute` instances by outputting, for each attribute, the `name`, (a whitespace,) a colon, (a whitespace,) the `name` of the `Type` instance as referenced by the `type` slot, and a carriage return. The `AI` rule is then accomplished, the AS rule ends by un-indenting the text and appending a "`}`" before a carriage return, and the `TT` rule is invoked again by the `TMI` rule, this time over the `User` instance a first time, and over the `String` instance in a second time. The final output is that one represented in Figure 5.

The process may be reverted by considering the text in Figure 5 and setting the generic analyzer and synthesizer in analysis mode to get the model. It is important to stress that the language definition in use is the same for both the analysis and the synthesis mode.

The `TM` start rule is a template for the `Model` metaclass, thus the analyzer creates a new `Model` instance that we will further call `m`. The `TMI` rule is then invoked, which iteratively calls the `TT` rule. The contextual instance for the `TM` and `TMI` rule is the newly created `m` instance. The `TT` rule will be called iteratively by the `TMI` rule with no contextual instance. The `TT` rule is a template rule for the `Type` metaclass; thus, the rule creates a new `Type` instance and sets it as the contextual instance. Then, the `TT` rule recognizes the '`Type`' keyword, an identifier ('`Mail`'), which will be placed in the `name` slot of the newly created contextual instance. Finally, the `TT` rule needs to perform a choice among two possibilities. As the contextual instance is under construction, the rule cannot rely on the guard constraints. In case of LL(k) parsing, as for the TCSSL tools, the choice will depend on the k next tokens; here, the next token is a '`{`', which will lead to choose the second alternative (i.e. we need $k \geq 1$). In case of backtrack parsing, as for the Sintaks tool, the first choice is taken, fails, and then the tool attempts the second choice. Note that if the first choice have had side effects (e.g. instantiation or slot update), they would have been all canceled. The second choice is a call to the `AS` rule. The `Mail` instance remains the contextual instance. The `AS` rule recognizes the '`{`' character, (it ignores the adornment), and calls the `AI` rule with `Mail` as the contextual instance, which creates the `From` and `To Attribute` instances. As the `AI` rule handles the `Type::attributes` property, the `From` and `To Attribute` instances are placed in the `attributes` slot of the `Mail Type` instance. Note that the template rule for the `Attribute` metaclass calls a sequence rule that contains a reference rule handling the `Attribute::type` property according to the `Type::name` property. A problem is that the `User Type` instance

is not created yet and thus could not be retrieved to be referenced by the `From` and `To` attributes. The referencing is then added to a "to-do" list, which contains all postponed tasks that are tried each time the model changes. So, the referencing of the `type` for the `From` and the `To` attributes will be done as soon as a `Type` instance will receive the `User` value in its name slot. If at the end of the analysis process the "to-do" list still contains tasks that cannot be performed (i.e. ending in error), the analysis ends in error. Once the `AS` rule has recognized the '`}`' character, the `TMI` rule appends the `Mail` instance (which is the contextual instance for the `TT` rule) to the children slot of `m`. The rest of the process behaves the same way by iterating again over the `TT` rule to successively recognize the `User` and `String` types.

5.2 Supplying TCSSL with a textual concrete syntax

We show here how to provide our meta-model for textual concrete syntax specification described in section 4 with a textual concrete syntax as that one shown in example in Figure 12. Note that this is only an example, and not an official concrete syntax that needs to be used with our tools or approach: one can very well supply another concrete syntax to specify his/her concrete syntaxes. However, for sake of brevity, we only show here the most interesting excerpt of the specification. Such an approach was validated for the two prototype tools.

The main rule is shown in Figure 13. A specification starts with the '`start`' keyword and continues an iteration to handle all the rules of the specification using the `RuleT` rule. For the first rule to be the start rule, we place a side-effect rule (in between `<?` and `>` tags) which tests whether the start rule is already there or not. In the latter case, the current rule, which must be the first rule to appear in the text, is referenced as the start rule. Remember that, as stated in section 4.10, a side-effect rule is only performed at analysis time.

```
start template handles TCSSpec do begin
      < "start" >
      iterate handles TCSSpec::rules do begin
            call RuleT
            <?    if self.start == void then
                  self.start := self.rules.one
            end >
            <@ "\n\n" >
      end
  end
```

Figure 13: Main TCSS rule for the TCSS language

Each instantiable concept shown in Figure 8 is supplied with a template rule. In parallel, each abstract concept is provided an alternative to choose among possible concretization. As an example, the Rule concept is represented by an alternative

(`RuleT`) that will propose choices among rules intended to manage sub-concepts. The `RuleT` rule is shown in Figure 14, and the template for the `Template` concept in Figure 15. Experience showed us that this repetition of generalization structure (as defined in the meta-model) in the alternatives of the concrete syntax is a common design pattern.

```
choice (RuleT)
      when self.oclIsKindOf(Template) do call TemplateT
      when self.oclIsKindOf(Iteration) do call IterT
      when self.oclIsKindOf(Alternative) do call AltT
      when self.oclIsKindOf(Sequence) do call SeqT
      when self.oclIsKindOf(Terminal) do call TerminalT
      when self.oclIsKindOf(Adornment) do call AdornT
      when self.oclIsKindOf(SideEffect) do call SET
      when self.oclIsKindOf(Value) do call ValueT
  end
```

Figure 14: The RuleT rule

```
template (TemplateT) handles Template do begin
      < "template" >
      call optionalId
      < "handles" >
      call MetaClassT
      < "do" >
      <@ "\n\t" >
      call RuleT handles Template::rule
      <@ "\b\n" >
  end
```

Figure 15: The template for the Template concept

6 Prototype implementations

Two prototypes have been implemented to validate our approach: Sintaks and TCSSL Tools. Both prototypes are operational and provide support for the model-driven definition of textual concrete syntax.

We first describe how the tools have been implemented, and then compare them to the above-presented approach.

6.1 Implementation

The *Sintaks* prototype is based on top-down recursive descent using a backtracking mechanism. It realizes both analysis and synthesis of concrete syntax by interpreting the textual concrete syntax specification provided as a model. It has been implemented on top of EMF in Eclipse.

We have not been trying to achieve high parsing performance; we have simply been investigating how modeling could be used to describe concrete syntax. We decided to implement Sintaks as a top-down recursive descent parser for sake of ease of implementation. This means that described grammars need to avoid left recursivity in order for the tool not to enter an infinite loop.

The main advantage of Sintaks is that one can rapidly prototype and test grammars, following the philosophy of the TXL parsing engine [41]. Conditions are formulated in a dedicated language able to test attribute values and object types. Side effect rules are not available in the current version.

TCSSL Tools were developed at CEA LIST and are a set of two different tools that make use of the same concrete syntax specification.

The first tool compiles a textual concrete syntax specification into a compiler compiler specification (namely ANTLR). The result is an LL(k) text processor that builds an EMF model from a textual representation. Here, in addition to avoid left recursivity, rules have to avoid LL(k) ambiguities. Actions and conditions are given using Java/EMF instructions.

The second tool compiles the same specification into a set of JET [46] code generation templates. Conditions are defined by an attribute value compared to an EMF Java expression. Those conditions are interpreted as attribute assignments at analysis time. Side effect rules are given using EMF Java instructions.

6.2 Major differences between the two prototypes

A first difference, which was experienced in TCSSL tools, is an automatic rule selection algorithm according to concept inheritance as described by the abstract syntax. Thus we avoid describing rules such as `RuleT` (see Figure 14), which are nothing but repetitions for inheritance hierarchies already provided by the meta-model. If this approach is more intuitive, a problem is that one cannot specify precedence among various template rules (e.g. multiplication over addition). Note that the language designer must always be aware of what is done implicitly by the generic analyzer and synthesizer generated by the TCSSL tools.

Another important difference is that TCSSL tools offer a more generalized analysis mechanism regarding template rules. Indeed, at analysis time, the concept instantiation can be made conditional, depending whether the instance under analysis can already be found in the model according to a Boolean query. In case the instance is found, it is used as the contextual object. In case no instance is found, language

engineers can state if the contextual instance should be created, or if the analysis process must end in error. This allows for example to overload the specification of the same element. One example is the C++ language in which the header and the implementation files for describing the same class are separated. If one needs to specify two different rules for the same meta-concept of C++ class, at analysis, only one rule is responsible for the instantiation of the concept, while the other merely finds it. This also allows specifying more complex search criterion than trying to match an identifier with an element in the model. A drawback of that method is that it is not compatible with a "to-do" list of the actions to be performed "as soon as possible". Indeed, search criteria can return different results depending on when they are executed during analysis.

7 Conclusion

The work presented in this paper provides a bi-directional bridge between texts and models. One can now think in terms of equivalence between those two worlds, and hence benefit from the best of both worlds. As a matter of example, model-transformation tools may be applied to texts (actually to their model counterpart), while pattern matching tools may be applied to models (actually to one of their textual representations).

More specifically, this paper addresses the issues of defining textual surface syntaxes on top of given metamodels. The advantage of our solution is that textual surface syntaxes can be fully customized and designed to be as specific as necessary. An important point is that the structure of abstract syntax (the metamodel) is completely independent from the structure of the concrete syntax.

We considered the two fundamental aspects of the problem: representing a model in text (synthesis) and constructing a model from text (analysis). Most traditional approaches make a clear distinction between those two cases, and propose two distinct solutions that have to be kept consistent. Instead, we propose a single specification for textual concrete syntax in which a generic text processor can find the necessary information to perform those two activities that are text analysis and synthesis.

Instead of inventing a brand new kind of specification, we have preferred to rely on well established principles of the grammarware community, especially on compiler compiler specification languages. However, two major improvements were necessary: the specification had to be made reversible and had to handle directly a predefined abstract syntax definition as given by a metamodel.

For the approach to be reversible, we provided (in plain English) different semantics for analysis and synthesis phases. For the approach to manipulate a predefined metamodel, we proposed concepts such as templates for meta-classes and

meta-features. Moreover, we made it possible to describe references, queries, and actions on the constructed/read model.

We built two different implementations to validate our proposal: Sintaks and TCSSL tools. Sintaks processes the specification by recursive descent implementing a backtracking mechanism. If this approach is recognized as not the most efficient, it is a nice mean to (partially) avoid ambiguities management while prototyping textual concrete syntaxes. The TCSSL tools transform the same specification into both a compiler compiler specification that includes directives directly on the constructed model, and into code generation templates that create formatted text depending on a model to be represented. As such, we directly benefit from compiler compiler advances regarding efficiency and scalability (at the necessary price of managing ambiguities).

In future developments we would like to take advantage of generalization hierarchies in metamodels to help in engineering concrete syntax specifications. Another possible improvement is to apply a bottom-up approach to parser technologies (LR) to handle a broader scope of possible grammars. Moreover, we would like to extend our approach in order to include an interactive synchronization between models and their textual representations. The result would be that models could be updated as soon as their textual representations are changed, and the reverse. To do so, a possible way is to build on graphical concrete syntax specification techniques or textual approaches to customize IDEs (such as in [36]).

Building this equivalence between texts and models is a cornerstone in the contexts of Model-Driven Engineering (MDE) and Domain-Specific Modeling (DSM). MDE promotes the usage of models as primary artifacts of software development projects, while DSM promotes the usage of the most suitable modeling language depending on the target domain and the considered level of abstraction.

In the MDE community, models are structured by their metamodels, which can be considered as defining the abstract syntax of the corresponding domain specific modeling language. To easier manipulate or understand models, one may want to provide a surface syntax for a modeling language. Ideally, it should be possible to have different concrete syntaxes, either graphical or textual, to represent a given model (i.e. a given metamodel instance) through different concrete syntaxes.

When used together with approaches for model executability (e.g. Kermeta [5]), our work offers a comprehensive mean to engineer textual domain-specific languages (DSLs) using a modeling paradigm. Indeed, one can now define the abstract syntax of his/her DSL by providing a metamodel, specify the operational semantics using Kermeta, and state one or more textual concrete syntaxes thanks to the TCSSL language described in this paper, all this in a model-driven continuum.

References

[1]. O.M.G. Object Management Group: Meta-Object Facility (MOF) Core Specification. (2006)

[2]. Eclipse: Eclipse Modeling Framework (EMF). (2005)

[3]. IBM: Emfatic. http://www.alphaworks.ibm.com/tech/emfatic

[4]. F. Jouault, J. Bézivin: KM3: A DSL for Metamodel Specification. The 8th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS), Bologna (2006)

[5]. Kermeta: The KerMeta Project Home Page. http://www.kermeta.org (2005)

[6]. OMG: Xml Metadata Interchange (XMI 2.1). (2005)

[7]. Sun_Microsystems: Metadata repository (MDR). (2005)

[8]. C. Atkinson, T. Kühne: The Role of Metamodeling in MDA. International Workshop in Software Model Engineering associated to UML '02, Dresden, Germany (2002)

[9]. J. Greenfield, K. Short, S. Cook, S. Kent: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Indianapolis (2004)

[10]. T. Clark, A. Evans, P. Sammut, J. Willans: Applied Metamodelling: A Foundation for Language Driven Development (2004)

[11]. OMG: MOF Model to Text Transformation Language (Request For Proposal). (2004)

[12]. J. de Lara, H. Vangheluwe: Using AToM3 as a meta-case tool. 4th International Conference on Enterprise Information Systems (ICEIS) (2002) 642-649

[13]. F. Fondement, T. Baar: Making Metamodels Aware of Concrete Syntax. European Conference on Model Driven Architecture (ECMDA), Vol. LNCS 3748 (2005) 190-204

[14]. andromda.org: AndroMDA.

[15]. T.J. Parr, R.W. Quong: ANTLR: A predicated-LL(k) parser generator. Software - Practice and Experience **25** (1995) 789-810

[16]. K. Czarnecki, S. Helsen: Feature-based survey of model transformation approaches. IBM Systems Journal **45** 621-645

[17]. P.-A. Muller, J.-M. Jézéquel: Model-driven generative approach for concrete syntax composition. Best Practices for Model Driven Software Development'04 (OOPSLA & GPCE Workshop), Vancouver (2004)

[18]. F. Fondement, R. Schnekenburger, S. Gérard, P.-A. Muller: Metamodel-Aware Textual Concrete Syntax Specification. (2006)

[19]. P.-A. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, J.-M. Jézéquel, D. Touzet: Sintaks. (2007) http://www.kermeta.org./sintaks

[20]. P.-A. Muller, F. Fleurey, J.-M. Jézéquel: Weaving executability into object-oriented meta-languages. MoDELS/UML 2005 - ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Vol. 3713. Springer, Montego Bay, Jamaica (2005) 264-278

[21]. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual: Centaur: the system. ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments, Vol. 13 (5) 14-24

[22]. N. Chomsky: Three models for the description of language. IRE Transactions on Information Theory (1956) 113-124

[23]. D.E. Knuth: Semantics of context-free languages. Mathematical Systems Theory **2** (1968) 127-145

[24]. J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger: Report on the algorithmic language ALGOL 60. Communications of the ACM **5** (1960) 299-314

[25]. ISO: ISO 14977 Information Technology - Syntactic Metalanguage - Extended BNF. (1996)

[26]. S.C. Johnson: Yacc: Yet another compiler compiler. UNIX Programmer's Manual, Vol. 2b (1979)

[27]. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: Compilers: Principles, techniques, and tools (2nd edition). Addison Wesley (2006)

[28]. T.W. Pratt: Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and System Sciences (JCSS) **5** (1971) 560-595

[29]. R. Heckel, H. Voigt: Model-Based Development of Executable Business Processes for Web Services. In: Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets. Springer, Berlin / Heidelberg (2003) 559-584

[30]. D. Hearnden, K. Raymond, J. Steel: Anti-Yacc: MOF-to-text. Enterprise Distributed Object Computing (EDOC) (2002) 200-211

[31]. OMG: Human-Usable Textual Notation. Object Management Group (2004)

[32]. I. Kurtev, M. Aksit, J. Bezivin: Technical Spaces: An Initial Appraisal. CoopIS, DOA´2002 Federated Conferences, Industrial track, Irvine (2002)

[33]. M. Wimmer, G. Kramler: Bridging Grammarware and Modelware. Workshop in Software Model Engineering associated to MoDELS'05, Montego Bay, Jamaica (2005)

[34]. M. Alanen, I. Porres: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Turku Centre for Computer Science (2003)

[35]. A. Kunert: Semi-Automatic Generation of Metamodels and Models from Grammars and Programs. Fifth International Workshop on Graph Transformation and Visual Modeling Techniques at ETAPS 2006 (2006)

[36]. M. Scheidgen: Textual Editing Framework (TEF). Humboldt-Universität zu Berlin (2007) http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html

[37]. F. Jouault, J. Bézivin, I. Kurtev: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. Generative Programming and Component Engineering (GPCE), Portland (2006) 249-254

[38]. P. Klint, R. Lämmel, C. Verhoef: Towards an engineering discipline for grammarware. ACM TOSEM **14** (2005) 331-380

[39]. P.-A. Muller, M. Hassenforder: HUTN as a Bridge between ModelWare and GrammarWare - An Experience Report. WISME 2005: 4th Workshop in Software Model Engineering (Satellite Event of MoDELS 2005), Montego Bay (2005)

[40]. D. Akehurst, P. Linington, O. Patrascoiu: OCL 2.0: Implementing the Standard. (2003)

[41]. J.R. Cordy: The TXL source transformation language. Sci. Comput. Program. **61** (2006) 190-210

[42]. D. Harel, B. Rumpe: Meaningful Modeling:What's the Semantics of "Semantics"? IEEE Computer **37** (2004) 64-72

[43]. N. Chomsky: Language And Mind. Cambridge University Press, Cambridge, USA (2006)

[44]. OMG: Unified Modeling Language: Superstructure Version 2.0. Vol. 2005 (2003)

[45]. M. Matula, S. Dedic: Java metamodel. Netbeans 4.0 (Sun Microsystems) (2005)

[46]. Eclipse: Java Emitter Templates (JET). (2005)