Model-Driven Analysis and Synthesis of Concrete Syntax

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

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

[2] Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
frederic.fondement@epfl.ch

[3] MIPS, Université de Haute-Alsace
Mulhouse, France
michel.hassenforder@uha.fr

[4] Laboratoire d'Intégration des Systèmes et des Technologies (LIST)
Commissariat à l'Energie Atomique (CEA)
Saclay, France
{remi.schneckenburger, sebastien.gerard} @cea.fr

Abstract. Metamodeling 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 concrete syntaxes with metamodels is still a challenge. 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 metamodels, 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 metamodels 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.

1 Introduction

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

Being able to parse a 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[8] or Xactium with XMF Mosaic[9] 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[10], is also being developed regarding concrete-to-abstract mapping. Although this paper focuses on textual concrete syntax, it is worth noticing that there are also ongoing researches about modeling graphical concrete syntax[11,12].

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 (such as Centaur[13]) that, when given the formal specification of a programming language (syntax and semantics), produce a language-specific environment.

There is currently a lot of interest in the modelware community about establishing bridges between so-called technological spaces[14]. For instance M. Wimmer and G. Krammler have presented a bridge from grammarware to modelware[15], whereas M. Alanen and I. Porres discuss the opposite bridge from modelware to grammarware, in the context of mapping MOF metamodels to context-free Grammars[16]. Kunert A. goes one step further and generates a model parser once the annotated grammar has been converted to a metamodel[17].

While a grammar could be considered as a metamodel, the inverse is not necessarily true, and an arbitrary metamodel cannot be transformed into a grammar[18]. Even metamodels dedicated to the modeling of an abstract syntax may require non-trivial transformations to target existing grammarware tools. In a previous work, we have experimented how to target existing compiler compilers to generate a parser for the generic HUTN language[19]. A similar experience, turning an OMG specification (OCL) into a grammar acceptable by a parser generator[20] 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 as two different topics. At this time, we are not aware of a model-based specification of concrete syntax that would allow both concrete-to-abstract and abstract-to-concrete mappings.

In this paper, we explore such a bidirectional mapping by defining a metamodel for the specification of textual concrete syntax in a context where abstract syntax is also represented by metamodels. 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.

The major difference with related works is that we do not try to build bridges between modelware and grammarware at the tool level. Actually, we are rather experimenting a new way of building tools for programming languages (such as compilers or IDEs) by using metamodels which embed results from the language theory directly into the modelware space.

This work takes place in the context of the Kermeta project[3]. Kermeta is an object-oriented executable meta-language, which can be used to specify both abstract syntax and operational semantics of a language.

This paper is organized as follows: the introduction examines some related works and motivates our proposal; section 2 presents our metamodel for concrete syntax, and explains the mechanics which are behind. Section 3 presents two examples which illustrate the way concrete syntax can be modeled and associated to models of the abstract syntax. Finally section 4 draws some general conclusions, and outlines future works.

2 Modeling concrete syntax

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

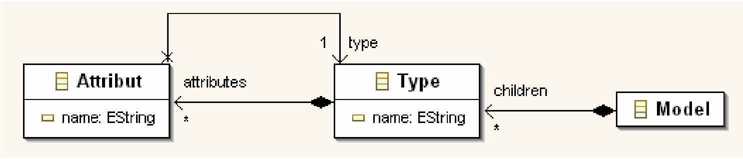


Figure 1: Metamodel of abstract syntax for a simple language

A typical concrete syntax may be:

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

Type User {
   Name : String
}

Type String;
```

Figure 2 : Example of concrete syntax

The metamodel on Figure 1 defines the abstract syntax (the concepts of the language), but nothing is said about concrete syntax. We have to find some way of specifying how a construction of the language is rendered in text. In the next sub-sections we will examine how a metamodel could be used for that purpose.

2.1 Overview of the metamodel

As seen in the previous example, the metamodel of the abstract syntax has to be complemented with concrete syntax information. In our case, this information will be defined in terms of another metamodel, which has to be used as a companion of the metamodel already defining the abstract syntax of the language under specification. This work is an evolution of our previous work which was limited to concrete syntax synthesis[21].

Figure 3 summarizes the approach. At runtime, both for analysis or synthesis, the models of abstract and concrete syntax are interpreted by a generic machine (written in terms of both metamodels) which performs the bidirectional transformation between texts and models.

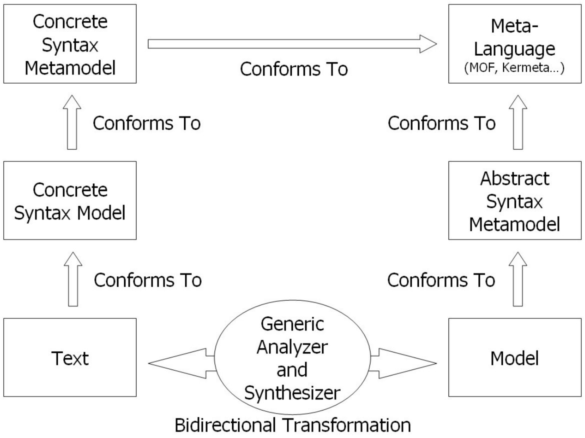

Concrete Syntax Metamodel
Conforms To
Meta-Language (MOF, Kermeta...)
Conforms To
Conforms To
Concrete Syntax Model
Abstract Syntax Metamodel
Conforms To
Conforms To
Text
Generic Analyzer and Synthesizer
Model
Bidirectional Transformation

Figure 3: A model-driven generic machine performs the bidirectional transformation

The metamodel for concrete syntax is displayed on Figure 4. A 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 metamodel of a language and the model of its concrete syntax is based on two meta-classes: *Class* and *Feature* respectively referencing the classes of the abstract syntax metamodel and their properties. The class *Template* makes the connection between a class of the metamodel and its corresponding rules. The class *Value* (and its sub-classes) and the class *Iteration* make the connection between the properties of a classes and their values. The class *Value* is used for properties whose multiplicity is 1 while the class *Iteration* handles collections associated to properties with multiplicity greater than 1. The remaining classes of the metamodel provide the usual construction for the specification of concrete syntax such as terminals, sequences and alternatives.

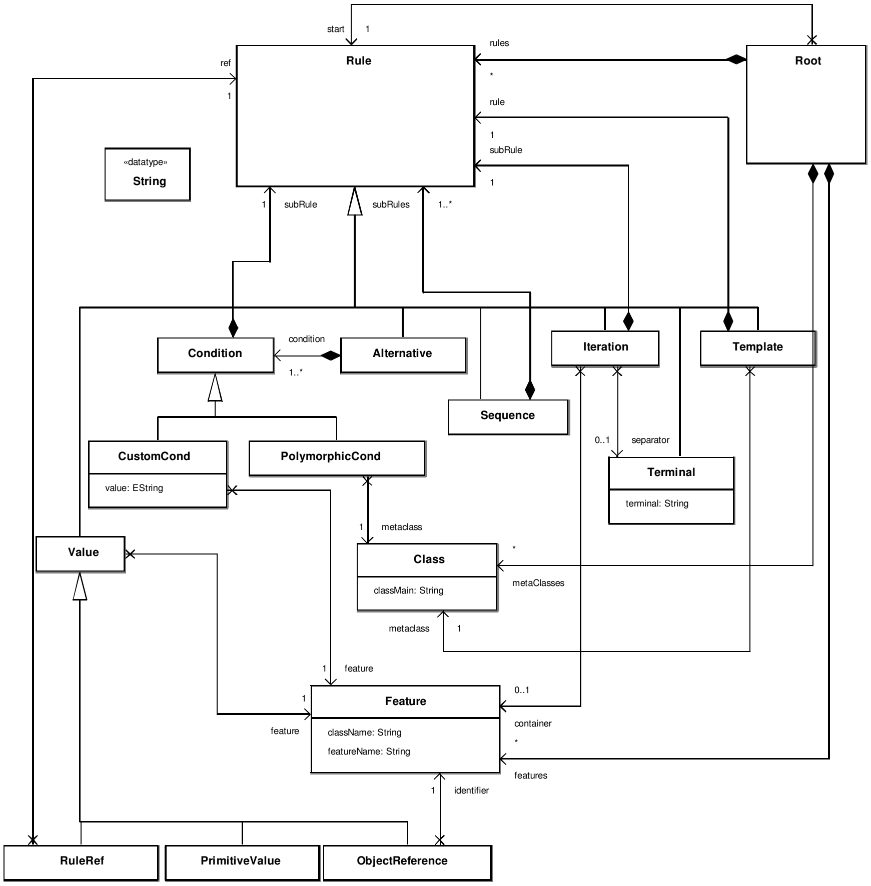


Figure 4: Overview of the metamodel for concrete syntax

During analysis, the input stream is tokenized, and parsed by a generic parser which operates by model-driven recursive descent. By model-driven recursive descent, we designate a recursive top-down parsing process which is taking advantage of the knowledge captured in the models of abstract and concrete syntax. While the parser recognizes valid sequences of tokens, it instantiates the abstract syntax, and builds an abstract representation (actually a model) corresponding to the input text.

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 metamodel) and uses the presentation information available in the concrete syntax model (conform to the concrete syntax metamodel) 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 is to perform two synthesis-parse sequences, and observe that there are no differences in both generated texts.

The following sub-sections detail the semantics associated to each elements of our concrete syntax metamodel, from both analysis and synthesis prospectives.

2.2 Template rule

A template rule makes the connection between a class of the metamodel (property *metaclass*) and a sub-rule.

- **Analysis semantics:** The template specifies that an object should be created. The metaclass 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.

2.3 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: T**he token in the input stream must be equal to the terminal value. The token is simply consumed. If the token does not correspond an exception is thrown.
- **Synthesis semantics:** The terminal value is appended to the output stream along with formatting information, such as white spaces.

2.4 Sequence rule

A sequence 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.

2.5 Iteration rule

An iteration specifies the repetition of a sub-rule an arbitrary number of times. This rule 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*).

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

2.6 Alternative rule

An alternative captures variations in the concrete syntax. An alternative has an ordered set of conditions which refers to a sub-rule. We have defined two types of conditions. Custom conditions are built over the features of a given class (may be a derived property, if the condition has to involve more than one class), while polymorphic conditions are built over the sub-classes of a given class.

- **Analysis semantics:** This is the most complex operation. Often there is no clue in the output stream to determine the condition (in the sense defined in the metamodel 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 solution (but also the most time consuming) is to try each branch of the alternative, until one does not fail. We have chosen to implement such backtracking algorithm in our prototype implementation. It is worth noticing that the ordered collection of conditions can also be used to handle priorities between conflicting sub-rules.
- **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.

2.7 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.

2.8 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 bear an attribute whose value contains such identifier.

- **Analysis semantics:** The reference which is extracted from the input stream is used as a key to query the model to see if it is possible 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) and creates a ghost to be referenced in place, and finally updates the element under construction with a reference to that ghost. By the end of the parsing process, all ghosts have to be resolved, unless there is a parsing error.
- **Synthesis semantics:** The identifier is directly printed to the output stream.

2.9 Rule reference rule

The rule *RuleReference* references a top-level rule fragment, 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.

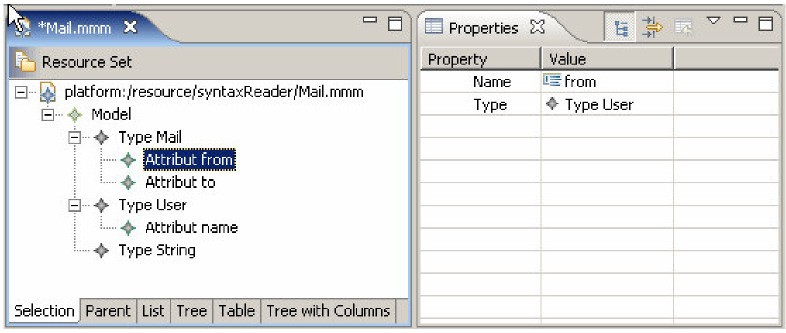
The following section shows how the concrete syntax metamodel is used for specifying concrete syntax.

3 Examples

The following examples have been implemented with our prototype.

3.1 A very simple example of concrete syntax specification

Going back to our small language example, we can use the reflexive editor of EMF (see Figure 5 below) to create a model directly as instances of the classes of the abstract syntax. This program defines three types (Mail, User and String).



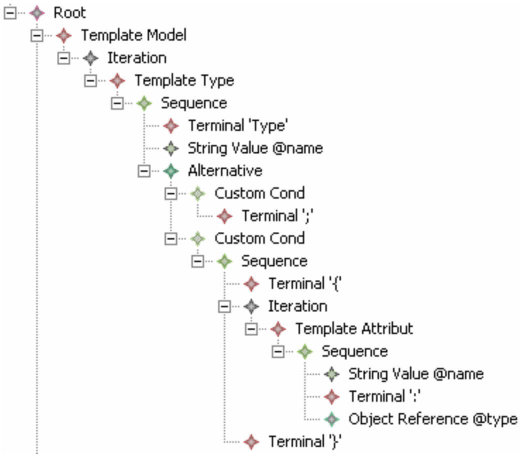
```
platform:/resource/syntaxReader/Mail.mmm
  Model
    Type Mail
      Attribut from
      Attribut to
    Type User
      Attribut name
    Type String
```

Property	Value
Name	from
Type	Type User

Figure 5: Use of the reflexive editor of EMF to create a model

We will now use our metamodel of concrete syntax to specify the textual representation. In the example of concrete syntax given earlier (see Figure 2), there is no specific materialization of the model in the text. A type is declared by a keyword followed by a name and an optional collection of attributes. A collection is denoted by curly braces; an empty collection is specified by a semi-column. Notice that the notation allows forward references to `User` and `String`.

Again, we may use the reflexive editor of EMF to instantiate the classes of the metamodel for concrete syntax. A straightforward model of this concrete syntax might be:

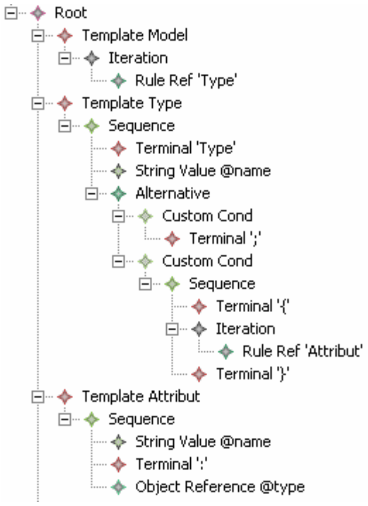


```
Root
  Template Model
    Iteration
      Template Type
        Sequence
          Terminal 'Type'
          String Value @name
          Alternative
            Custom Cond
              Terminal ';'
            Custom Cond
              Sequence
                Terminal '{'
                Iteration
                  Template Attribut
                    Sequence
                      String Value @name
                      Terminal ':'
                      Object Reference @type
                Terminal '}'
```

Figure 6: Use of the reflexive editor to model concrete syntax

In this model, there is only one top-level rule which describes the concrete syntax of the language. The model is built as a cascade of rules. The model starts with an iteration over types. The sequence explains that types start with the terminal '`Type`', followed by a name, and then an alternative because types may have a collection of attributes. Attributes when present are delimited by curly braces. The collection of attributes is expressed by an iteration, which in turn contains a sequence made of a name, followed by a separator (terminal '`:`') and finally a reference to a type.

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 variation, for the same concrete syntax. Links between independently defined templates are realized with rule references (`RuleRef`).



```
Root
  Template Model
    Iteration
      Rule Ref 'Type'
  Template Type
    Sequence
      Terminal 'Type'
      String Value @name
      Alternative
        Custom Cond
          Terminal ';'
        Custom Cond
          Sequence
            Terminal '{'
            Iteration
              Rule Ref 'Attribut'
            Terminal '}'
  Template Attribut
    Sequence
      String Value @name
      Terminal ':'
      Object Reference @type
```

Figure 7: Variation with top-level reusable templates

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

3.2 Modeling simple arithmetic expressions

This second example is based on the traditional example of arithmetic expressions as found in many textbooks about compilation[22]. The following picture represents the metamodel (the abstract syntax) for simple arithmetic expressions.

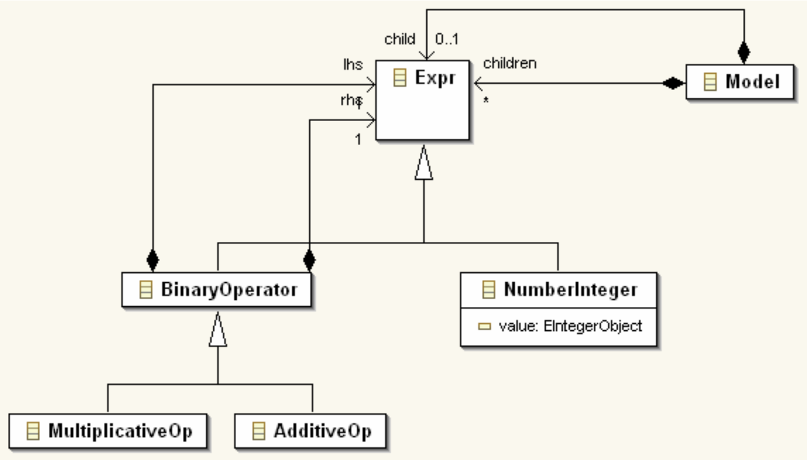
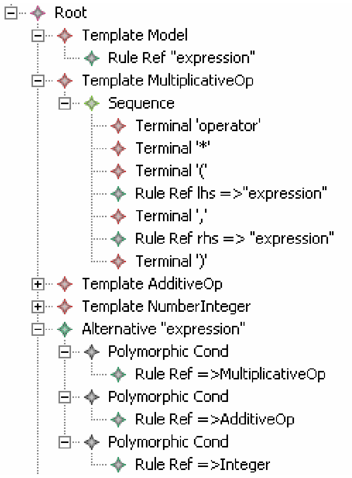


Figure 8: Metamodel for simple arithmetic expressions

The abstract syntax contains the following elements:

- **Model**. Represents the root of any arithmetic expression.
- **NumberInteger**. Represents an integer.
- **MultiplicativeOp**. Represents a binary multiplication operator.
- **AdditiveOp**. Represents a binary addition operator.

Let's first examine a prefixed concrete syntax for expression. The operator is always located in front of the operands, and the priorities are implicitly expressed at the invocation of each operator. The following concrete syntax model addresses such prefixed notation.



```
Root
  Template Model
    Rule Ref "expression"
  Template MultiplicativeOp
    Sequence
      Terminal 'operator'
      Terminal '*'
      Terminal '('
      Rule Ref lhs =>"expression"
      Terminal ','
      Rule Ref rhs => "expression"
      Terminal ')'
  Template AdditiveOp
  Template NumberInteger
  Alternative "expression"
    Polymorphic Cond
      Rule Ref =>MultiplicativeOp
    Polymorphic Cond
      Rule Ref =>AdditiveOp
    Polymorphic Cond
      Rule Ref =>Integer
```

Figure 9: Concrete syntax for prefixed expressions

Notice that we have here several independent top-level rule fragments. The first fragment is the entry point template. The second fragment (`MultiplicativeOp`) is in charge of multiplication operators. It is built as a sequence made of the keyword '`operator`' followed by a star sign and (between parentheses) two consecutive invocations of the expression rule, to handle respectively the left-hand-sign (lhs) and right-hand-side (rhs) of the expression (separated by a comma). The third fragment (`AdditiveOp`) is built on the same scheme as the precedent fragment. The (`NumberInteger`) models simple integer values. The last rule fragment (the expression alternative) states which branch to take based on the actual class of the parsed element. The order of these alternatives is not meaningful for prefixed expressions.

Non-factorized alternatives are a typical issue for parsers which operate by recursive descent. With our approach, a non-factorized grammar (such as the repetition of '`operator`' in both multiplicative and additive operators) is not really an issue, because the strategy used to analyze an alternative is based on backtracking. Branches are tried in a row, until there is no parsing error, which means that the correct branch has been found. Practically speaking, this process remains reasonably quick, as it is possible to validate (or not) the current branch as soon as the symbol of the operator is encountered in the input stream.

A typical concrete syntax example might be:

```
operator + ( operator * ( 3 , 2 ) , 1 )
```

While such prefixed notation is easy to parse by machines, humans tend to prefer an infixed representation, closer to the way computation is done manually. The same expression represented in infix notation is:

```
3*2+1
```

Now it is not possible anymore to ignore the operator precedence, as it was the case with the non-ambiguous prefixed notation. The usual approach consists in encoding the priority of operators by introducing new intermediate symbols such as `Term` and `Factor`.

In line with parsers which operate by recursive descent, our prototype implementation requires also converting rules with left-recursivity into rules with right-recursivity. In the end, the grammar (expressed in BNF) becomes:

```
<Expression>::= <AdditiveOP> | <Term>

<Term> ::= <MultiplicativeOp> | <Factor>
```

```
<Factor> ::=<NumberInteger>

<AdditiveOp> ::=<Term> ‘+’ <Expr>

<MultiplicativeOp> ::= <Factor> ‘*’ <Term>

<NumberInteger> ::= [0-9]+
```

Such rewriting may be a little bit tedious, but is only required for languages which support infix notation for arithmetic expressions.

The following picture shows how this reformulation is presented in a model conform to our concrete syntax metamodel.

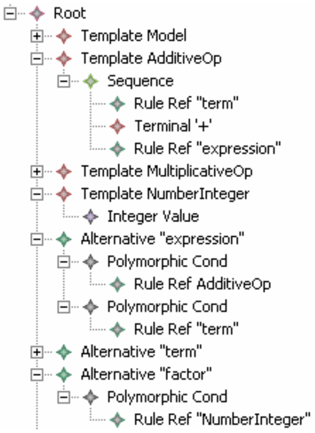


Figure 10: Concrete syntax for infixed expressions

The template `AdditiveOp` defines a sequence which starts by invoking the `term` rule, follows by a plus sign and finishes by a call to the `expression` rule. The `MultiplicativeOp` is built on the same scheme, while the `NumberInteger` template handles Integer values. As described earlier in BNF, the expression alternative is made of either a call to the `AdditiveOp` rule or a call to the `term` rule.

4 Conclusion

This work may be viewed as an experimentation for the specification of concrete syntax in the context of meta-modeling applied to language engineering.

We have proposed a new approach, based on metamodels, which supports a formal bidirectional mapping of both concrete-to-abstract, and abstract-to-concrete syntax.

A prototype, based on recursive descent, which realizes both analysis and synthesis of concrete syntax has been implemented on top of EMF in Eclipse. This prototype has been used to parse and pretty-print several DSLs, as well as the examples presented in this paper.

Our work is obviously far from bringing definitive answers to the complex problems of applying metamodels to language engineering but, along with the capabilities of executable meta-languages such as Kermeta, it suggests that languages can be fully specified in terms of metamodels, and that tools can be automatically derived from these metamodels to support these languages.

In the short future, we will be investigating how to avoid rewriting rules, potentially by using mechanisms similar to those behind LL (*) engines such as found in ANTLR v3[23]. We are also working on a graphical editor (based on templates), to make the specification of the concrete syntax even more intuitive for non-specialists.

A lot of work is still beyond us to make tools based on this approach as robust and efficient as the one in the grammarware space. However, the presented material may contribute, with many other ongoing research works, to a better understanding of metamodeling applied to language engineering.

References

[1] OMG, *Meta-Object Facility (MOF) 1.4*, OMG Document formal/02-04-03 (2002).

[2] IBM, *Emfatic*, http://www.alphaworks.ibm.com/tech/emfatic

[3] Muller, P.-A., F. Fleurey and J.-M. Jézéquel, *Weaving executability into object-oriented meta-languages*, in: International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005), pp. 264–278.

[4] OMG, *Xml Metadata Interchange (XMI 2.1)*, OMG Document formal/05-09-01 (2005).

[5] Sun Microsystems, *Metadata repository (MDR),* (2005), http://mdr.netbeans.org/

[6] Eclipse, *Eclipse Modeling Framework (EMF),* (2005) http://www.eclipse.org/emf/

[7] Atkinson, C. and Kuehne T., *The role of meta-modeling in MDA*, in: Workshop in Software Model Engineering (WISME@UML), Dresden, Germany, 2002.

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

[9] Clark, T., Evans A., Sammut P. and Willans J., *Applied metamodelling: A foundation for language-driven development* (2005). URL http://albini.xactium.com

[10] OMG, *MOF Model to Text Transformation Language* (Request For Proposal), OMG Document ad/2004-04-07 (2004).

[11] de Lara, J. and Vangheluwe H., *Using AToM3 as a meta-case tool*, in: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS), 2002, pp. 642–649.

[12] Fondement, F. and Baar T., *Making Metamodels Aware of Concrete Syntax*, in: European Conference on Model Driven Architecture (ECMDA), LNCS 3748 (2005), pp. 190–204.

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

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

[15] Wimmer, M., Kramler, G., *Bridging Grammarware and Modelware*, WISME Workshop, MODELS / UML'2005, October 2005, Montego Bay, Jamaica.

[16] Alanen, M., and Porres, I., *A Relation Between Context-Free Grammars andMeta Object Facility Metamodels*. Technical report, Turku Centre for Computer Science, 2003.

[17] Kunert A., *Semi-Automatic Generation of Metamodels and Models from Grammars and Programs*, in Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques at ETAPS 2006, april 2006.

[18] Klint P., Lämmel R., and Verhoef C., *Towards an engineering discipline for grammarware*. ACM TOSEM, Vol. 14, N. 3, PP 331-380, May 2005.

[19] Muller, P.-A., Hassenforder M., *HUTN as a Bridge between ModelWare and GrammarWare – An Experience Report*, WISME Workshop, MODELS / UML'2005, **October** 2005, Montego Bay, Jamaica.

[20] OMG. UML 2.0 Object Constraint Language (OCL) Final Adopted specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, 2003.

[21] Muller, P.-A., P. Studer and J.-M. Jézéquel, *Model-driven generative approach for concrete syntax composition*, in: Workshop in Best Practices for Model Driven Software Development, Vancouver, Canada, 2004.

[22] Aho A.V., Sethi R., Ullman J., D., *Compilers, Techniques and Tools*, Addison Wesley. 1986

[23] Parr, T., *Another Tool for Language Recognition* (ANTLR) (2005), http://www.antlr.org/