

# **A Relation Between Context-Free Grammars and Meta Object Facility Metamodels**

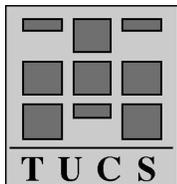
**Marcus Alanen and Ivan Porres**

TUCS Turku Center for Computer Science

Åbo Akademi University

Lemminkäinsenkatu 14A, FIN-20520 Turku, Finland

e-mail: name.surname@abo.fi



**Turku Centre for Computer Science**

**TUCS Technical Report No 606**

**March 2003**

**ISBN 952-12-1337-X**

**ISSN 1239-1891**

## **Abstract**

Metamodels present the language of models, much in the way that grammars present the language of programs. In this paper, we study the relation between context-free (Backus-Naur Form) grammars and Meta Object Facility metamodels and identify when and how we can convert a grammar to a metamodel and a metamodel to a grammar. An example of this mapping for a subset of Java is shown.

**Keywords:** BNF, MOF, Metamodelling, Transformations

**TUCS Laboratory**  
Software Construction Laboratory

# 1 Introduction

In this paper, we study how to bridge the gap between program code and other text artifacts whose structure is defined using context-free grammars (Backus-Naur Form, BNF) and software models whose structure is defined using the Meta Object Facility (MOF).

The relation between a metamodel and a BNF grammar can in practise be defined using two mappings, one transforming a BNF grammar to a MOF metamodel, and one transforming a MOF metamodel to a BNF grammar. We would like to be define under which conditions these mappings are possible, and under which conditions they are not. Furthermore, the transformations would need to be correct, deterministic and reversible.

Figure 1 shows a concrete example using UML as the user's primary metamodel and Java as the target programming language. The user's actual model and metamodel are assumed to be transformed via a series of mappings to the desired output metamodel, and from there to a BNF syntax tree, from which program code can be written. The lower layer in the Figure with models and syntax trees are commonly known as the M1 layer in the modelling community, and the upper layer is known as the M2 layer. Objects at the M1 layer are instances of objects at the M2 layer.

The mapping from a grammar to a metamodel is called  $G \rightarrow MM$  while the mapping from a program text (actually a parsed syntax tree representing the program text) is called  $T \rightarrow M$ . The process is reversible, e.g. we can transform a Java model into a Java program, using the mappings  $G \leftarrow MM$  and  $T \leftarrow M$ . Therefore, we write  $G \leftrightarrow MM$  and  $T \leftrightarrow M$  to describe these two relations. The main contents of this article are devoted to describing the relation  $G \leftrightarrow MM$  and how to automate it.

The metamodels representing the grammars, i.e. the Java metamodel in the Figure, do not necessarily resemble the UML metamodel. However, both the metamodel representing the grammar and the UML metamodel are based on MOF. Therefore it is possible to define a mapping from one metamodel to another using model transformation technology.

Actually, this is the most important consequence of the work presented in this article. Code generation and reverse engineering becomes an issue of model-to-model transformations from the source metamodel to the BNF-specific metamodel. These transformation can be describing using the upcoming QVT standard [5]. The actual transformations are specific to each pair of programming language and modelling language.

We proceed as follows. After an overview of related work, Section 2 defines the domain of BNF grammars and metamodels more formally. Section 3 defines the relation between BNF grammars and metamodels, as well as the relation between BNF syntax trees and actual models. We look at an example in Section 4 and provide a conclusion with ideas for future work in Section 5.

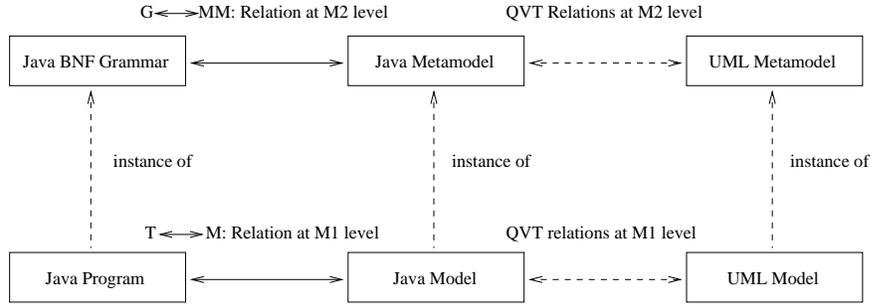


Figure 1: An example of using the Java programming language. The paper presents the algorithms for  $G \leftrightarrow MM$ .

## 1.1 Related Work

Generating metamodels from other already-existing descriptions is not a new idea, and the relation between BNF grammars and MOF metamodels has been discussed before. The SODIUS company informs [7] to have “developed a number of technologies to derive metamodels from various sources such as BNF grammars, COM API DLLs, UML models” [9], albeit nothing is said how this has been done. The ADORA [11] framework maps graphic elements to textual ones, defining a BNF grammar. However, the transformation is created manually with a lot of work to be done, e.g. every metamodel element must be mapped to BNF explicitly. Furthermore, there is no way to transform a BNF to a metamodel since ADORA lacks a meta-metamodel.

An approach taken by Varró and Pataricza in [10] is to create a simple meta-model for all (Chomsky) grammars, and not use any specific metamodel depending on the target programming language. The benefit is that several algorithms become easier, while the disadvantage in our opinion is that it is also possible to abuse the grammar metamodel to create invalid models, requiring verification with the target context-free grammar in any case. We would like to lift the rules of the target context-free grammar to the modelling domain, instead of keeping it separate.

## 2 Definitions

In this section we define the two domains more formally. This is required for us to be able to define the relation and finally the algorithms that implement the relation.

### 2.1 Backus-Naur Form

Backus-Naur was developed by John Backus and Peter Naur in the end of the 1950’s for the ALGOL 60 programming language [1]. It has since then been improved into what is commonly referred to as Extended BNF, of which there

are several kinds. Among such extensions, ABNF [2] is an Internet RFC and ISO/IEC 14977:1996(E) defines a common uniform precise EBNF syntax. All common extensions are for practical purposes only, as the various EBNF grammars can also be represented as BNF grammars. Thus, any algorithms based on EBNF could work with only BNF, even though using EBNF is in many cases more practical. The extensions provided are to more clearly convey the structure of e.g. a programming language.

Formally, BNF grammars consists of a set of tokens, divided into terminals and non-terminals. Furthermore, there is a set of *production rules* mapping a non-terminal  $N$  to a sequence (ordered bag) of tokens. Grouping can be expressed using parentheses ( ) and choices in a group are separated by a vertical bar |. Then, a *parse tree* can be generated by starting with an initial *start token* and applying any rules until only terminals are left. Such a parse tree can then be used to write the concrete syntax, e.g., a Java grammar can be used to write parse trees which are then transformed into textual programs. We require the start token to be a non-terminal.

Common extensions are optional tokens using brackets [ ] or a question mark ?, any amount of repetition (zero or more) using the Kleene star operator \* and a repetition of one or more using the Kleene cross operator +. The extensions are specialisations of the  $\{m,n\}$  operator which denotes repetition of the previous group or token from  $m$  up to  $n$  times, inclusive. Thus, optionality is  $\{0,1\}$ , the Kleene star is  $\{0,\infty\}$ , and the Kleene cross is  $\{1,\infty\}$ . This version of Extended BNF is similar to what many others have already defined. Due to the practical usefulness of EBNF we have chosen to use it as defined in this subsection, instead of plain BNF.

## 2.2 Metamodel

Metamodels have primarily been developed and advertised by the Object Management Group with its MOF [4, 8] standard. Unfortunately no standard formal definition has been written on metamodels for the purposes of this article. We define a metamodel  $M$  to consist of named (meta)classes  $C$  of which one is called the *root*, generalisations  $G$  and named enumerations  $E$ . The generalisations form a class hierarchy by mapping a subclass to its superclasses,  $C \rightarrow C^*$ , forming a cycle-free graph. An instantiation of a metaclass is called a model element, and its *base type* is the metaclass. Its *type* is the metaclass or any superclass or the metaclass, transitively. Every metaclass  $C$  can define a set of *properties*. The effective set of properties of a metaclass is those defined by itself or transitively by any of its superclasses. The names of the properties in the effective set must be unique. The properties define the features or *slots* of the model elements, by describing what kinds of connections there can exist between model elements.

A property  $P$  consists of a name, multiplicity range, various characteristics and target metaclass or enumeration. The name is for convenience to make it easier to distinguish properties, multiplicity ranges define how many connections to in-

stances of the target the slot (instantiated property) should have to be well-formed. Common multiplicity ranges are 0..1 for an optional and 1..1 for an obligatory connection, 0..\* for any amount of connections, 1..\* for at least one connection, all very similar to EBNF repetition ranges. A property has also several other characteristics: its elements can be in order (*ordered*), can occur multiple times (*bag*) or it can denote ownership (*composition*). Two properties can be connected to form a meta-association between two metaclasses in the metamodel.

Every enumeration contains a list of values. Instances of an enumeration then must contain one of the values, implying that there must be at least one value defined. Usually, the individual values are described with strings. Enumerations contain no properties and have no sub- or superclasses.

As such, our metamodels are a subset of OMG MOF metamodels, especially with respect to new property characteristics in MOF 2.0/UML 2.0 such as subsets, derived unions and overriding. As of yet, no mathematical definitions of these characteristics have been given and they are not taken into account in this article.

### 3 $G \leftrightarrow MM$ : A Relation at the Metamodel Level

In this section we identify our initial requirements for bridging EBNF and metamodels, and present two algorithms for creating a mapping between these two domains. We will use the vocabulary given in the QVT proposal: transformations can be split into two separate subtypes, *relations* and *mappings*. Relations are “multi-directional nonexecutable transformation specifications”, whereas mappings are “transformation implementations, potentially uni-directional.” As such, our relation is called  $G \leftrightarrow MM$  and it can be implemented using two mappings,  $G \rightarrow MM$  and  $G \leftarrow MM$ , the first mapping from grammars to metamodels and the second from metamodels to grammars.

There are several important considerations when trying to build a relation between EBNF grammars and MOF metamodels. For the actual mappings, two algorithms need to be created that operate on the M2 level; one that takes a EBNF grammar as input and produces a metamodel as output, and one that takes a metamodel as input and produces a EBNF grammar as output. The desired qualities of the relation are listed below.

1. Every valid grammar can be mapped to a valid metamodel.
2. Every valid metamodel can be mapped to a valid grammar.
3. The relation is reversible, i.e. the generated output can be mapped to the initial input.
4. Valid models should produce valid syntax trees, and vice versa.
5. All information should be contained in the grammar or metamodel. No external data or dependency, apart from the generic algorithms, should be required.

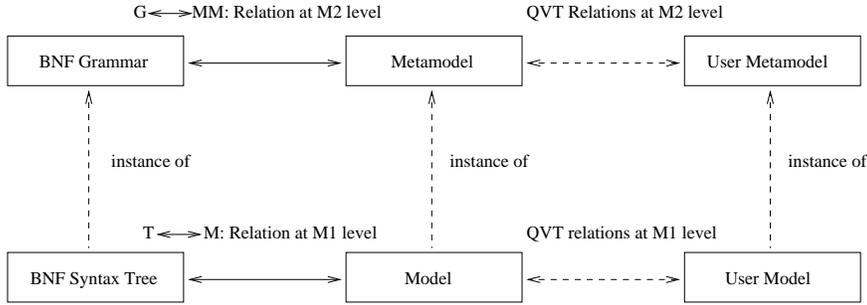


Figure 2: The relation  $G \leftrightarrow MM$  described at the M2 level between BNF grammars and OMG metamodels implies another relation  $T \leftrightarrow M$  at the M1 level between syntax trees and models.

For the definitions of a valid grammar or metamodel in requirements 1 and 2, we refer to the descriptions given in Section 2. The validity is important for the mapping to work in practise on any input. In other words, we would like that  $mm = G \rightarrow MM(g)$  produces a valid metamodel and  $g = G \leftarrow MM(mm)$  produces a valid grammar, preferably for arbitrary valid input parameters for a metamodel  $mm$  or a grammar  $g$ . For reversibility,  $mm = G \rightarrow MM(G \leftarrow MM(mm))$  and  $g = G \leftarrow MM(G \rightarrow MM(g))$  is required. Using the two algorithms at the M2 level, a relation  $T \leftrightarrow M$  at the M1 level can be defined which maps EBNF syntax trees to models, and vice versa. However, we do not dwell on any algorithms at the M1 level, as they more or less follow from the definitions at the M2 level. We discuss the usability of the algorithms using an example of a subset of the Java BNF in Section 4.

A concrete example of this relation was given in the introduction in Figure 1. Figure 2 shows a generalisation of this and shows how our work fits into the bigger scheme of software development using models.

### 3.1 $G \rightarrow MM$ : Mapping From an EBNF Grammar to a Metamodel

In this subsection we describe an algorithm for transforming EBNF grammars to metamodels. Without loss of generality, we assume that every nonterminal occurs only once on the left-hand side of the production rules. If this is not the case, we can collect two or more rules  $N \Rightarrow R_1, N \Rightarrow R_2, \dots$  into one rule  $N \Rightarrow (R_1|R_2|\dots)$ .

The general idea of the algorithm is the following: there is a metaclass matching every nonterminal or group of choices. Then, the sequence of tokens is mapped to ordered unidirectional composite properties, one property for each token. The order between the properties is kept by naming the properties in an alphabetically rising order. Trivially, multiplicities can be mapped directly. For groups, the individual choices in a group subclass the metaclass that defines the choice. For terminals, we create an enumeration  $e$  with only one value, equal to the terminal's

string value. Then, a property is created whose target is  $e$ . The slots will then contain values from that enumeration. Obviously, those values can only be the single defined value, and thus we have “forced” a valid model to include specific terminal strings. In practise these forced enumerations instances could be automated and hidden in a modelling tool.

One quickly notices that some common EBNF notations could be transformed more conveniently. Often, a EBNF grammar gives a choice between static strings, such as  $N \Rightarrow (\mathbf{public}|\mathbf{protected}|\mathbf{private})$ . Rather than creating a cumbersome metamodel with many metaclasses, one single enumeration with three values is sufficient. Also, typically we would also special-case e.g. words and numbers, and form instances of the primitive data types **string** and **integer**, respectively.

The algorithm is presented in pseudocode in Figure 3. The **namegen** function which returns lexically increasing names for properties is worth of special mentioning. It is noteworthy that the generated metamodel would have been quite different if only plain BNF would have been used. Also, we have used only a subset of the capabilities of metamodels and as shall see in the next subsection, this also leads to complications.

### 3.2 $G \leftarrow MM$ : Mapping From a Metamodel to an EBNF Grammar

Describing a mapping from metamodels to EBNF grammars is in many ways more demanding than the opposite. The reason is that metamodels inherently contain more information than EBNF grammars. While a EBNF grammar is quite similar in that it can be presented as a graph of nodes and directed edges, the edges themselves do not contain as much information as properties in a metamodel. EBNF forms a tree; metamodels form graphs with special edges that can be interpreted in many ways.

This fundamental difference does not mean that EBNF could not be used to map arbitrary metamodels to EBNF. For example, XML Metadata Interchange (XMI) [6] could essentially be defined using EBNF and contain an arbitrary metamodel. The problem lies in reversibility; an EBNF grammar mapped to a metamodel and back to a grammar ought to result in the same EBNF grammar, otherwise no relation truly exists. However, the issue does mean that a metamodel can express the structure of information more conveniently than an EBNF grammar. The versatility of metamodels has increased in successive versions of MOF, and so with time this gap is widening even further. Furthermore, EBNF imposes additional restrictions on the metamodel. Since tokens are described in a sequence, the elements in the slots are always ordered. No non-composite associations can be transformed, since EBNF is a tree, not a general graph. As such, there is no direct isomorphism between any metaclass in an arbitrary metamodel and a token in a grammar.

The problem is the opposite of the problem in the previous subsection, and they are related. A solution would be to somehow infer structure from BNF production rules to all metamodel concepts, thereby making it easier to map all the metamodel

```

Function add_properties( $E, s$ )
  For  $t$  in  $s$ ,
    Create a new property  $p$ 
     $p.isComposition := true$ 
     $p.isOrdered := true$ 
     $p.multiplicity := t.multiplicity$ 
     $p.name = namegen()$ 
    If  $t.token$  is a group,
       $a := new\_group(t.token.choices)$ 
    Else if  $t.token$  is a nonterminal,
       $a := new\_element(t.token)$ 
    Else if  $t.token$  is a terminal,
      (special-case for primitive types here)
      If no enumeration for  $t.token$  has been created,
        Create a new enumeration  $a$  with a single value, the name of  $t.token$ 
      Else
         $a :=$  Previously created enumeration for  $t.token$ 
    Set the target of  $p$  to  $a$ 
    Add the property  $p$  to  $E$ 
Function new_group( $c$ )
  Create an abstract metaclass  $A$ 
  For every sequence  $s$  in the choices  $c$ 
    Create a metaclass  $E$  and add  $A$  as its superclass
    add_properties( $E, s$ )
  Return  $A$ 
Function create( $T$ )
  Return previously created element for  $T$  if it exists
  Special-case: if  $T$  represents an enumeration,
    Create a new enumeration  $E$  for  $T$  with the name of  $T$ 
    Populate  $E$  with the values from  $T$ 
  Otherwise,
    Create new metaclass  $E$  for  $T$  with the name of  $T$ 
    Find the production rule  $T \Rightarrow s$ 
    add_properties( $E, s$ )
  Return  $E$ 
Function grammar_to_metamodel( $B$ )
  Create new metamodel  $M$ 
   $M.root := create(B.start\_token)$ 
  Return  $M$ 

```

Figure 3: An algorithm for mapping an arbitrary EBNF grammar  $B$  into a metamodel  $M$ .

concepts back to BNF production rules.

In any case, reversibility of the algorithm presented in the previous section is possible, and it is presented in pseudocode in Figure 4. Here the importance of keeping the properties in order is revealed. Without a predefined order on the naming or an external information source that sorts the properties, the token sequence generated would end up in arbitrary order, with ensuing problems generating or reading program code.

### 3.3 Relation at the Model Level

As we have defined both unidirectional mappings that comprise a relation at the M2 layer, it is possible to push them to the M1 layer to form a new relation  $T \leftrightarrow M$ . This relation defines mappings from a BNF syntax tree (i.e., a program) to a model, and from a model to a syntax tree. Naturally the BNF grammar and the metamodel must be related according to the previously mentioned definitions, and they must also both be known when mapping at the M1 level.

Creating this relation requires some compiler knowledge, especially if the algorithms are to be efficient. For syntax trees, we assume a thorough knowledge of which token in the syntax tree correspond to which token in the BNF grammar, and at what production rule. Usually syntax trees already contain this information. We also assume that the implementation can keep track of which metamodel elements and properties are related with which tokens in the BNF grammar. As the general idea of the mappings is quite straightforward, we will not present detailed algorithms. However, the outlines of the algorithms are presented next.

#### 3.3.1 Syntax Tree to Model

Mapping a syntax tree to a model is an instantiation of the metamodel elements according to our mapping from EBNF to metamodels. For every nonterminal, a model element is created, and for every group, a model element is created depending on the choice that has been made in the group by the syntax tree. The slots of the elements are filled in alphabetical order, according to what tokens are in the tree for that nonterminal or group. Exactly how these decisions are determined is beyond the scope of this paper, but we note that usually syntax trees do contain the necessary information.

#### 3.3.2 Model to Syntax Tree

Mapping a valid model to a syntax tree is a straightforward traversal of the elements of the root element of the model. For this, we require that composition in properties denotes ownership and thus forms a tree at the model level. This means that e.g. depth-first traversal of a model is trivial, and guaranteed to visit each element exactly once. Similarly to the mapping from syntax trees to models, each model element corresponds to a nonterminal or group choice in the BNF grammar,

```

Function create(E):
  Return previously created element for E if it exists
  If E is an enumeration,
    If E only has one value,
      Create a terminal T for E with the name of E
      Return T
    Else
      Create a nonterminal T for E with the name of E
      Add the production rule  $T \Rightarrow (v_1, v_2, \dots)$ ,
        where each  $v_i$  is an enumeration value from E
      Return T
  Else if E is of primitive type,
    Return special token depending on E
  Else if E is a metaclass,
    If E is abstract,
      Create a new group g for E
      For every subclass S of E,
        Add create(S) to g as a choice
      Return g
    Else if E has superclasses,
      Create a new sequence of tokens s for E
       $n := \text{nil}$ 
    Else
      Create a new nonterminal n for E
      Create a new production rule  $n \Rightarrow s$ , where s is a sequence of tokens
    For every property p in E in alphabetical order,
      if p.isComposition,
         $a := \text{create}(p.\text{target})$ 
        Add a new token a to s, with multiplicity given by p
    If  $n \neq \text{nil}$ ,
      Return n
    Return s

Function metamodel_to_grammar(M)
  Create new BNF grammar B
   $B.\text{start\_token} = \text{create}(M.\text{root})$ 
  return B

```

Figure 4: An algorithm for mapping a metamodel *M* meeting certain criteria into a BNF grammar *B*.

and from this the syntax tree can be built. Primitive types and enumeration values map to terminal tokens. The slots are inspected in the same alphabetical order as the properties were generated.

## 4 Example

In this section we present a simple example based on the grammar for the Java programming language [3]. A basic BNF syntax has been modified to take into account the extended BNF, mainly using the Kleene star repetition \*. The example EBNF, shown in Figure 5 can describe a package construct with import statements and class declarations. Classes can only contain method declarations, and their body is just one big string.

The generated metamodel is shown in Figure 6, and it can be translated back without loss by the algorithms given in this paper. We have used a special token **String** in the grammar which is translated to a primitive type **string** in the metamodel. Similarly, one can define special tokens for e.g. integers or floating-point values. Notable is the seemingly odd way to subclass using just one class (e.g. **C13** and **C14**), resulting from a grouping with only one choice. The output of the algorithm is intentional, as we would like a change in the BNF to disturb already-existing models as little as possible. The assumption is that since a group existed, more choices might appear in it later, in which case we would have to subclass in any case.

There is an immediate consideration regarding the naming of the properties and metaclasses. The naming is crucial since the alphabetical order of the properties are used to re-create the BNF grammar. However, the names are rather inconvenient for real-world use when the metamodel or grammar changes. A separate naming of properties and book-keeping of the correct order would remove this obstacle, but would require us to keep external information about the BNF grammar and metamodel. However, developments in either grammar or metamodel could be accompanied with suitable transformation rules for old syntax trees or models.

## 5 Conclusions

We have presented a generic approach to the problem of converting program text into models. It is based on two algorithms that describe a relation from grammars to metamodels. We show how to map any BNF grammar into a metamodel and how to map a restricted class of metamodels into BNF grammars.

These algorithms do not map any arbitrary grammar into any arbitrary metamodel. For example, they cannot be used to map the Java grammar into the UML metamodel. If we want to solve this problem, we can still use the work presented in this paper to map the Java grammar into what can be called the Java metamodel. Due to these algorithms this step is automatic. Once we have obtained the Java metamodel we can use any of the novel model-to-model transformation approaches

```

goal := package_declaration ? import_declaration * class_declaration *
package_declaration := package String ;
import_declaration := import String .* ? ;
class_declaration := class_modifier * class String super? class_body
class_modifier := ( public | abstract | final )
super := extends String
class_body := { method_declaration + }
method_declaration := method_header method_body
method_header := method_modifier + result_type method_declarator
method_modifier := ( public | protected | private |
                    static | abstract | final |
                    synchronized | native )
result_type := ( String | void )
method_declarator := String ( ( String ( , String ) * ) ? ) ;
method_body := { String }

```

Figure 5: An example of a subset of the Java grammar. Applying the  $G \rightarrow MM$  mapping produces the metamodel in Figure 6, and it can be converted back to this same grammar without loss.

to define the mapping between the Java metamodel and the UML metamodel. The same reasoning applies in the reverse direction when we want to transform a UML model into a Java program. In this sense, this work solve the first and last step on a round-trip engineering tool.

It is noteworthy that we do not concern ourselves with how a valid character stream is mapped to a specific BNF grammar or how the BNF syntax tree can be mapped to a character stream. The first task can be accomplished by a lexical analyser while the second task is the work of a pretty printer that introduces whitespace between tokens following the conventions of the programming language. These tasks are well studied in the compiler literature.

There is still room for improvement in our work. There are still some MOF metamodels that cannot be mapped into BNF grammars. The reverse problem is equivalent in nature; our mapping from grammar to metamodel must also be able to construct any metamodel and all the facilities provided by metamodels. It is clear that some constructs from BNF can be mapped to subparts of a metamodel using more viable mechanisms, e.g. the special-casing of choices with only terminals ( $T_1|T_2|\dots$ ) into an enumeration, or the usage of primitive datatypes. The question is if more common structures of BNF grammars can in practise be mapped to complex metamodel structures, and the other way around.

However, the inability to perform this task automatically might be an inherent limitation due to the differences of structure between BNF and metamodels. The issue could perhaps be sidestepped if we were to annotate BNF tokens and parts of production rules somehow, so we could convey more meaning of the grammar

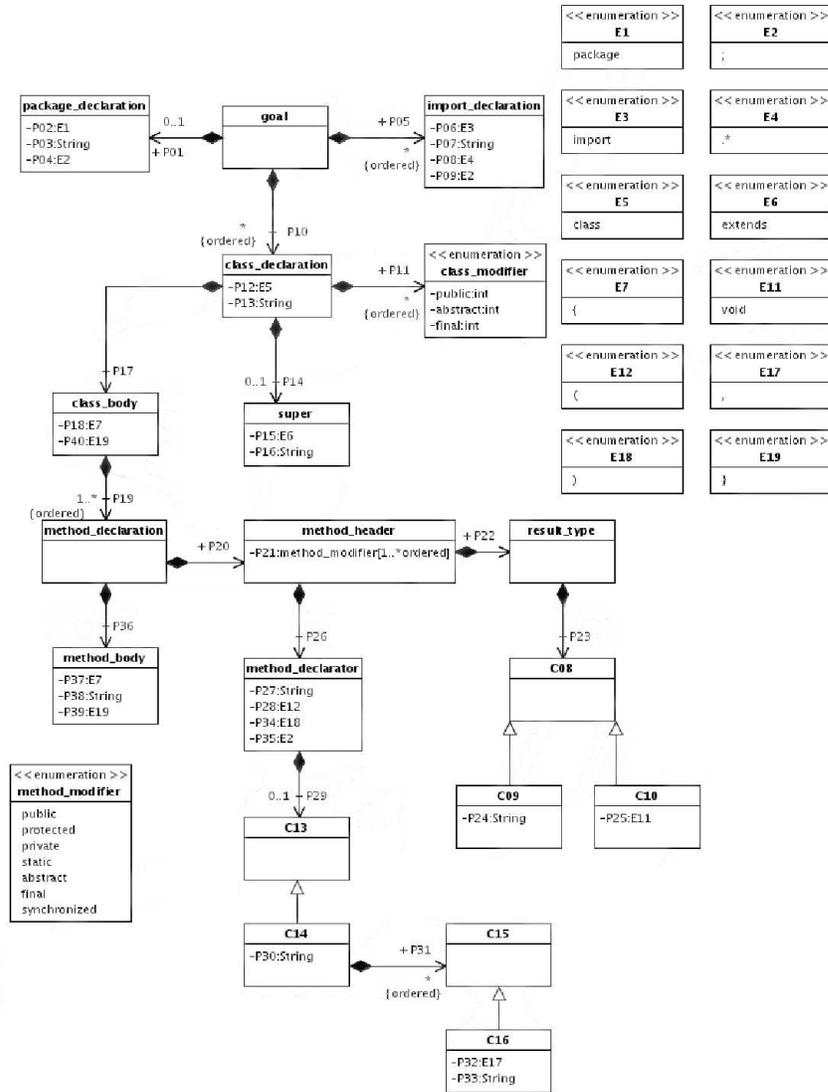


Figure 6: The generated metamodel of the BNF in Figure 5 (all superclasses are abstract). Applying the  $G \leftarrow MM$  mapping produces the original grammar.

to the algorithms. It is of practical interest to consider how much manual work is required for annotated BNF grammars, and what benefit they bring.

## References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, January 1986.
- [2] P. Overell D. Crocker. Augmented BNF for Syntax Specifications: ABNF — RFC 2234, November 1997. Available at <http://www.ietf.org/rfc/rfc2234.txt>.
- [3] Sun Microsystems. Java Syntax Specification.
- [4] OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03, available at <http://www.omg.org/>.
- [5] OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at [www.omg.org](http://www.omg.org), 2002.
- [6] OMG. XML Metadata Interchange, version 1.2, January 2002. Available at <http://www.omg.org/>.
- [7] OMG. *MDA-components: Is there a Need and a Market?* June 2003. Document ad/03-06-01.
- [8] OMG. Meta Object Facility, version 2.0, April 2003. Document ad/03-04-07, available at <http://www.omg.org/>.
- [9] Sodius. <http://www.sodius.com/>.
- [10] Dániel Varró and András Pataricza. Mathematical Model Transformations for System Verification. Technical report, Budapest University of Technology and Economics, May 2001.
- [11] Yong Xia and Martin Glinz. Rigorous EBNF-based Definition for a Graphic Modeling Language. In *Proceedings of APSEC 2003, the 10th Asia-Pacific Software Engineering Conference*. IEEE Computer Society Press, August 2003.

**Turku Centre for Computer Science**  
**Lemminkäisenkatu 14**  
**FIN-20520 Turku**  
**Finland**

<http://www.tucs.fi>



**University of Turku**

- **Department of Information Technology**
- **Department of Mathematical Sciences**



**Åbo Akademi University**

- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**



**Turku School of Economics and Business Administration**

- **Institute of Information Systems Science**