Bridging Grammarware and Modelware

Manuel Wimmer and Gerhard Kramler

Business Informatics Group, Vienna University of Technology, Austria
lastname@big.tuwien.ac.at

Abstract. In Software Engineering many text-based languages and supporting tools are used, forming the grammarware technical space. Currently model driven engineering is the new emerging paradigm for software engineering, which uses modelling languages and tools, forming the modelware technical space. Transition to the new technical space and interoperability between these two technical spaces is needed in many development scenarios. Building a bridge between these two technical spaces is a tedious task, that has to be repeated for each language to be transformed. Therefore, we propose a generic bridge between grammarware and modelware technical spaces, that can generate a specific bridge based on the EBNF of a given language semi-automatically. The generation comprises of two steps, (1) automatic generation of metamodel corresponding to the EBNF and (2) annotations to provide the additional semantics not captured by the EBNF. The generated bridge is capable of bi-directional transformations between sentences (programs) and corresponding models and can be used in re-engineering applications and for integration of text-based and model-based tools.

1 Indroduction

The term *technical spaces* was introduced in [7] and with it the demand for bridges between several technical spaces. Manual bridging of technical spaces is a error prone and recurring task, typically relevant in model driven engineering but also in other software engineering disciplines, e.g., the migration from relational databases to XML based documents. Therefore a need for tools arises, which support and automate interoperability between technical spaces.

A bridge between grammarware and modelware is useful in many software development tasks. Not only forward engineering but also reverse engineering of existing software systems is a suitable field of application. Regarding the latter the *Object Management Group (OMG)* is working on model-based reverse engineering and software modernization. For that purpose a special work group for *Architecture-Driven Modernization* (*ADM*, [9]) has been initiated. The main target of ADM is to rebuild existing applications, e.g, legacy systems, as models and then perform refactorings or transform them to new target architectures. A bridge between grammarware and modelware can act as a basic infrastructure tool to support various ADM tasks.

Bridging two technical spaces involves several tasks, such as processing the artifacts in the source technical space and transforming them into new artifacts,

J.-M. Bruel (Ed.): MoDELS 2005 Workshops, LNCS 3844, pp. 159–168, 2006.

that can be processed by tools from the target technical space. Currently transformation definitions are placed at the M2 level, e.g., between Java grammar and UML class diagram metamodel. This approach has a huge drawback, because for each pair of languages exactly one bridge must be defined. For another pair a different bridge has to be developed. Generally speaking for technical space 1, which has n languages, and technical space 2, which has m languages, $n * m$ bridges are needed. Considering the huge amount of languages in the grammarware, bridging at the M2 level is not a satisfying solution. In modelware there are not existing as many languages as in grammarware existing, but it seems that many Domain Specific Modeling Languages [3] will be developed in the near future. The exploding number of language combinations between these two spaces requires a more generic approach, which allows to generate bridges for all language combinations in the same way automatically.

In this paper we propose a generic mechanism for the semi-automatic generation of a specific bridge between grammarware and modelware based on the EBNF of a given language. EBNF [13] [4] is the most used metalanguage for defining programming languages in the grammarware. However other forms of metalanguages [6] are common, and sometimes there are no formal grammar definitions available at all. It is important to note that our work does not address these problems - for more information see [5]. In the modelware our mechanism is based on MOF [8], which is the main standard for defining metamodels. The first step in the proposed process is the production of a metamodel and a transformation program for transforming programs into models. The resulting metamodels and models have some drawbacks, because of the genericity of the transformation rules. In order to eliminate these unintentional properties, we introduce a second step - the optimization of metamodels and models. Some optimization steps can be done automatically - we call this process *condensation* - but some optimizations have to be done semi-automatically by user-annotations. This optimization process is called *customization*. With manual annotations it is possible to add semantics to the metamodels that are not captured by the original grammars.

The rest of the paper uses Mini-Java [11] as an running example and is structured as follows. Section 2 provides an overview of the framework architecture. Section 3 presents details of the parsing process and the raw meta model. Furthermore mappings between EBNF and MOF concepts are discussed. In section 4 details about the optimization steps in the condensation process are shown. Section 5 represents the main features of the customization process, in particular the manual annotations. Section 6 gives an overview of the related work and how it differs from this work. Finally, section 7 draws some conclusions and outlines future work regarding implementation and application.

2 Overview of the Framework Architecture

Our proposed framework exploits the fact, that grammarware and modelware have metalanguages. The main idea is to find correspondences between EBNF and MOF concepts and to use these correspondences for defining bridges.

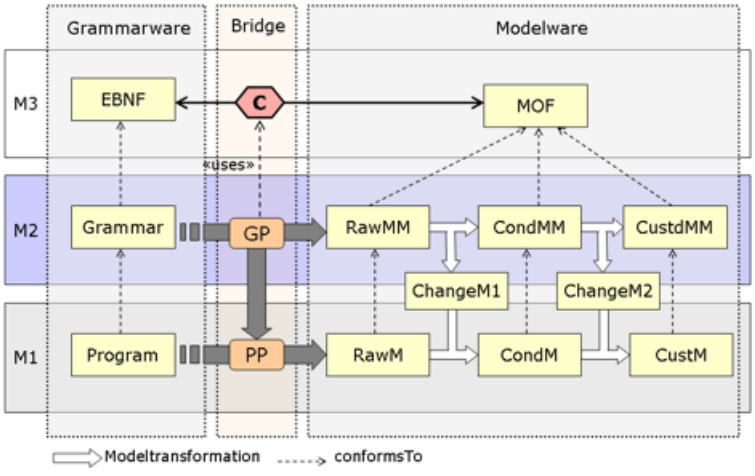


Fig. 1. Framework Overview

Figure 1 shows the main idea by a correspondence relation in the M3 layer between EBNF and MOF. EBNF is a reflexive language, i.e., EBNF can be described in EBNF. We utilize this property for constructing an attributed grammar, which defines a grammar for EBNF and implements the correspondences between EBNF and MOF as transformation rules. A compiler-compiler takes the attributed grammar as input and generates a parser called *Grammar Parser (GP)*. On the one hand the GP converts grammars defined in EBNF into *Raw Metamodels* and on the other hand it generates a parser for programs, which are conform to the processed grammar. This parser is called *Program Parser (PP)*. Via the PP programs can be transformed in a *Raw Models*. The Raw Metamodel and the Raw Model are expressed in *XML Metadata Interchange* (*XMI* [10]), thus they can be processed by modelware tools. The grammar parser and the program parser act as the main bridging technologies between grammarware and modelware. It is important to note, that both parsers are automatically generated from grammars and from the correspondences between EBNF and MOF.

Once the Raw Metamodel and the Raw Model are created, we have reached the modelware technical space. However the Raw Metamodel and the Raw Model can grow very big in terms of number of classes. To eliminate this drawback, some transformation rules for optimization are introduced, which can be automatically executed by model transformation engines. The optimization rules are applied to the Raw Metamodel and the outcome of this transformation is called *Condensation Metamodel*. Not only the metamodel has to be optimized, but also the model has to be adjusted in such a way that it is conform to the Condensation metamodel. This adjustment is defined by a *Change Model*, which includes all required information to rebuild the Raw Model as a *Condensation Model*.

Furthermore our approach provides a mechanism to add additional semantics to the metamodel that cannot be expressed in EBNF. These additional semantics

are attached to the Condensation Metamodel by manual annotations. In particular, this annotations cover aspects like Identification/References semantics, data types and improved readability. The annotated Condensation Metamodel is automatically transformed into a *Customized Metamodel*. Again, the changes in the metamodel layer must be propagated to the model layer. For this task we introduce a second *Change Model*. The Change Model covers all user-defined modifications and propagates them to the condensation model, which is finally transformed into a *Customized Model*.

The main reason why the optimizations are done in the modelware and not in the grammarware is that the framework is aligned to be used by model engineers. Apart from that two further reasons have influenced our design decision: (1) the optimization rules require potentially complete parse trees, which are available by the Raw (Meta)models. (2) MOF, in contrast to EBNF, has an inherent annotation mechanism, therefore we decided not to directly annotate the EBNF grammars.

3 Parsing and Raw (Meta)model

The main target of the parsing process is to transform the textual definitions of the grammarware into a format, that can be processed by model engineering tools. The first step is parsing the grammar of a given language. For our framework we decided to use EBNF as metalanguage, because it is the most used metalanguage to define grammars. In order to process each EBNF grammar identical, the syntax of the grammars must conform to the standardized EBNF syntax [4]. The Grammar Parser for processing the EBNF grammars can be generated by a compiler-compiler and an attributed grammar. The attributed grammar contains the structure and the concepts of EBNF, as well as method calls for the transformation of EBNF concepts to MOF concepts.

For the transformation of EBNF concepts into MOF concepts, the correspondences between these two metalanguages have to be clarified. This has been done in previous work - see [1], where relations between EBNF and MOF concepts are discussed. Based on this work, we constructed a complete set of transformation rules, which are summarized in the following. The transformation rules are organized along the major EBNF concepts, i.e., production rule, non-terminal, terminal, sequence, repetition, optional and alternative.

Rule 1: Represent every left hand side of a production rule as a class. The elements of the right hand side are represented as classes as defined by the following rules and are connected to the left hand side class by a containment association. An exceptional case is the first production rule of the grammar. In this case the LHS class is additionally marked with the ≪start_symbol≫ stereotype.

Rule 2: Represent every non-terminal as a class, which is called like the non-terminal name plus _REF and is marked with a ≪reference≫ stereotype. The class is connected to the corresponding left hand side class of the non-terminal by an association.

Rule 3: Represent every terminal as a class named as T plus a consecutive number and marked with the stereotype ≪terminal≫. The value of the terminal is represented as the value of the literal property.

Rule 4: Represent a sequence as an anonymous class called SEQ plus a consecutive number and marked with ≪sequence≫ stereotype. The classes representing the sequence are attached to the anonymous class by a containment association. The associations are assigned with an ≪ordered≫ stereotype and a key/value pair indicating the position of the element in the sequence.

Rule 5: Represent every repetition by an anonymous class called REP plus a consecutive number and marked with a ≪repetition≫ stereotype. The anonymous class has a one-to-many association with the class representing the repeated element. Note, that it is important to tag the association end with multiplicity many as ordered. This constraint is required to rebuild the linear order of EBNF.

Rule 6: Represent every option by an anonymous class called OPT plus a consecutive number and marked with a ≪option≫ stereotype. The anonymous class has a zero-to-one association with the class representing the optional element.

Rule 7: Represents every alternative as a subclass of an anonymous class called ALT plus a consecutive number. The anonymous class is marked with an ≪alternative≫ stereotype and is defined as an abstract class.

Mappings for the grouping concept can be ignored in the transformation process, because a group can be directly transformed in a sequence element or in an alternative element. Furthermore there is no need for a special transformation rule for recursive definitions, because this rule can be constructed from the combination of rule 2 and 7.

The Grammar Parser implements the listed transformation rules and is therefore able to generate a Raw Metamodel expressed in XMI from a grammar expressed in EBNF. Not only a Raw Metamodel is produced by the Grammar Parser, but also a Program Parser is derived from the correspondences at M3 level and the generated Raw Metamodel. The Program Parser creates a corresponding model representation in XMI - called Raw Model - from textual-based programs.

4 Condensation

The generated Raw Metamodels and Raw Models have some unintentional properties. The generic transformation rules, explained in chapter 3, let the design size of the models grow immoderate, because lots of anonymous classes were introduced. To eliminate this drawback, transformation rules for optimization of the Raw Metamodel and of the Raw Model are established, which can be executed by model transformation engines automatically. The optimization rules

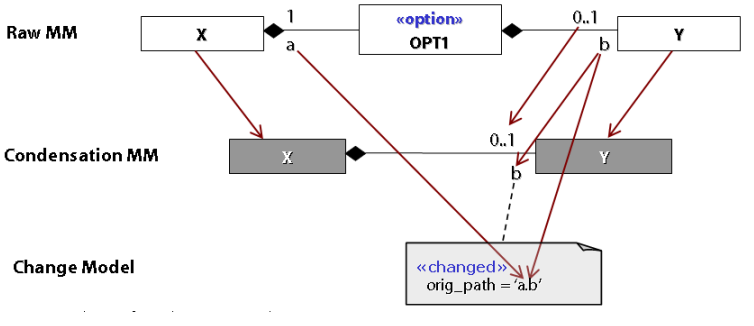
have to derive (1) a optimized Metamodel - called *Condensation Metamodel* - from the Raw Metamodel and (2) a *Change Model*, which includes all necessary informations to rebuild a Condensation Model from a Raw Model. From the combination of Condensation Metamodel and Change Model it is possible to rebuild a model from a Raw Model, so that it is conform to the Condensation Metamodel. The Change Model is included in the Condensation Metamodel as special marked annotations.

The optimization of the Raw Metamodel starts with the upper class in the class hierarchy - the class marked with ≪start_symbol≫. From this class the optimization rules are executed in depth-first order, because the containment hierarchy of the Raw Metamodel corresponds to the tree strucutre of the EBNF grammar and therefore can be processed like a tree. When an optimization rule matches for a given class, the children of this class must be checked, whether optimization rules can be applied on them. This has to be done recursively until a child class is found on the path, for which no optimization rules can be applied. Then all optimization rules are performed on the path in reverse order, until the class with the original match is reached. From this class the depth-first optimization is continued. It is important that the changes in the child classes are reflected in the optimization of the upper classes, because this makes it possible to execute the optimization in one step and no temporary metamodels are needed. In the following the optimization rule 1 for the Raw Metamodel is shortly quoted. The complete description of rule 2 (optimization of sequences), 3 (optimization of terminal classes) and 4 (optimization of alternatives) may be found in [12].

Rule1: Deletion of anonymous classes for options and repetitions, where precondition 1 holds. The child class in the containment hierarchy takes the place of the deleted class. The original path of the child class must be saved in the Change Model.

Precondition:
(1) The anonymous class owns only one child class and the type of the child class is either sequence or non-terminal.

Effect and Change Model for Precondition (1) shown by option-elimination:



The order in which the rules are applied, must comply with the listing order of the rules. The rules do not change the semantics of the metamodel, only some anonymous classes are eliminated or some terminal classes are restructured in a more compact way. The expressiveness of the language is the same as with the un-optimized metamodel, but the size of the metamodel and of the model is reduced.

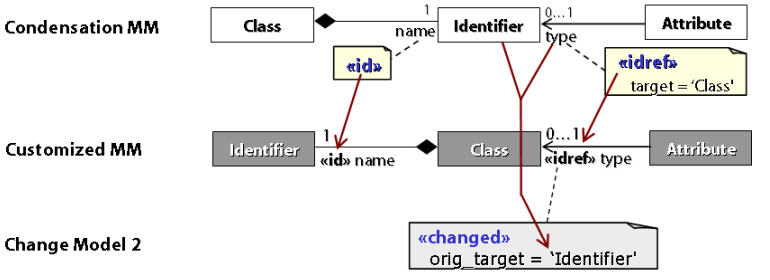
5 Customization

The aim of the generated metamodels and models is to maximize the understandability of languages and program specifications. As far as we have described, the metamodels and models are only graphical representations for the textual definitions. In this section we introduce a semi-automated mechanism to add additional semantics to the automatically generated metamodels, which cannot be expressed in EBNF. The user can attach annotations of a pre-defined set to the Condensation Metamodel. In order to enhance the quality of the generated metamodel by supporting improved readability, identification/reference semantics and data types, we propose the following manual annotations for the Condensation Metamodel. From these annotations it is possible to derive a Change Model to propagate the user-defined changes in the metamodel to model level in order to rebuild a Customized Model from the Condensation Model. The Change Model is again included in the generated Metamodel, in this case in the Customized Metamodel. The complete description of the annotation mechanism for data types and readability may be found in [12].

Identification/Reference: In metamodels the differentiation between Identification and Reference can be achieved easily. In contrast, grammars have no appropriate concepts for describing Identification or Reference in order to give a clue for the language designer's intentions. In our Mini-Java example the problem reveals with the class *Identifier*. An Identifier can be a variable, a class or a method name. If a variable is of type of a certain class, the identifier must be a class name and not a variable or method name. To indicate this constraint, we need additional information from the user in form of annotations, because this information cannot be derived from EBNF grammars.

Annotations: For this aspect two annotations are available (1) the ID annotation for defining an Identification and (2) the IDREF annotation for defining a reference. The stereotype ≪id≫ should be assigned to associations, which connect the element with the identifier class. The association end must have a multiplicity of 1. Also a new reference association to the actual referenced class is needed. This is done by marking existing associations, which should be redirected, with the stereotype ≪idref≫ and define the new target association end. The transformation has to delete the pseudo references, which become obsolete by the use of the user-created reference associations. The Change Model stores the original target of the reference association.

Effect and Change Model:



Readability: In metamodels the ability to give convenient names to elements makes it easier for the user to understand the intentions of the language designer. With annotations for names we allow the user to replace anonymous names, like OPT1 or ALT1 with convenient labels.

Data Types: MOF provides the following data types for metamodelling: string, integer and boolean. In contrast, EBNF has no concepts for data types and so they have to be described with complex expressions. These expressions result in complex class structures in the generated Condensation Metamodel. The substitution of such complex structures with data types provided by MOF leads to a more convenient metamodel.

As an example for the final output of our framework see Figure 2. In this figure an excerpt from the Customized Metamodel for Mini-Java is shown. Due to lack of space, we ignore stereotypes and tagged values concerning the concrete syntax. Note that the metamodel is completely automatically generated with our framework, except for manual annotations in the Condensation Metamodel. On the basis of Figure 2 it is readily identifiable, that the proposed optimization rules and annotations lead to a very intuitive metamodel.

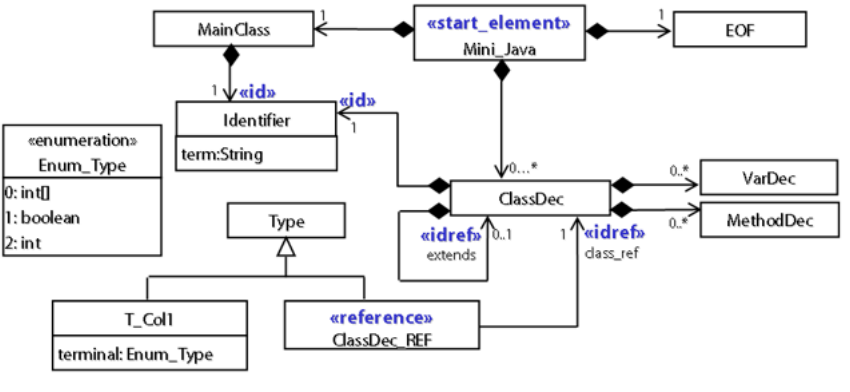


Fig. 2. Core of the Mini-Java Metamodel

6 Related Work

This section compares our work with related research activities: on the one hand approaches addressing the bridging of technical spaces and on the other hand approaches discussing the mapping between EBNF and MOF concepts.

Another mapping approach based on the M3-layer is described in [2]. In particular this work is focused on bridging model engineering and ontology engineering. Unlike our approach, only languages are transformed, which are based on MOF and have an XMI representation. Therefore it is possible to transform the XML representation with XSLT.

Alanen and Porres [1] discuss relations between context-free grammars and MOF metamodels. In contrast to our work, they only define mappings for M2 based on M3. We extend this approach by mapping not only M2, but also M1 based on M3. Furthermore, we establish various optimizations in order to get a more user-friendly metamodel. Our approach has used some mapping concepts of this previous work to define the grammar to raw metamodel transformation rules.

7 Conclusion and Future Work

In this work we have presented a generic framework, which supports the transformation of grammars into metamodels and of programs into models. The used transformation process is based on the M3 level, which allows to generate bridges between grammarware and modelware automatically. Furthermore we have described how to build a minimal and user-friendly metamodel through a number of optimization rules and user-annotations.

We are currently working on a prototype for the presented framework. As soon as our prototype is full functioning, we will use it to evaluate our framework with larger grammars and extensive programs. We hope this facility brings more insight on bridging grammarware and modelware. Therefore our next steps will be searching for additional optimization rules and user annotations, which allow a more flexible design mechanism for the final metamodel.

References

1. Marcus Alanen and Ivan Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, Turku Centre for Computer Science, 2003.
2. J. Bézivin, V. Devedzic, D. Djuric, J.M. Favreau, D. Gasevic, and F. Jouault. An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In *Proceedings of the first International Conference on Interoperability of Enteprise Software and Applications, (INTEROP-ESA 05)*, 2005.
3. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.
4. ISO. ISO/IEC 14977:1996(E), Information technology - Syntactic metalanguage - Extended BNF, 1996.

5. P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM TOSEM*, May30 2005. To appear; Online since July 2003, 47 pages.
6. K. Koskimies. Object Orientation in Attribute Grammars. *LNCS, vol. 545. Springer-Verlag*, pages 297–329, 1991.
7. Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technical Spaces: An Initial Appraisal. *CoopIS, DOA2002 Federated Conferences, Industrial track, Irvine*, 2002.
8. OMG. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/ptc/03-10-04.pdf, 2004.
9. OMG. Architecture Driven Modernization. www.omg.org/adm, 2005.
10. OMG. *XML Metadata Interchange (XMI) Specification*. OMG, http://www.omg.org/docs/formal/05-05-01.pdf, 2005.
11. Ryan Stansifer. EBNF Grammar for Mini-Java. http://www.cs.fit.edu/~ryan/cse4251/mini_java_grammar.html, August 2005.
12. Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. Technical report, Vienna University of Technology, http://www.big.tuwien.ac.at/research/publications/2005/1105.pdf, 2005.
13. Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM, 20(11)*, November 1997.