Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 211 (2008) 111–119

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Semi-automatic Generation of Metamodels and Models From Grammars and Programs

Andreas Kunert [1]

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany

Abstract

Most recent languages used in the field of computer science (programming languages, modelling languages, . . .) are defined by using a grammar-based notation. Although the definition of a language by metamodels is more convenient in terms of understandibility, precision and the ability to reuse abstract concepts from other language definitions, most current textual languages are still missing a complete metamodel. Unfortunately this implies that modern model-based software development tools are not able to process programs written in those languages.
We propose a framework which generates a metamodel for each programming language defined by a grammar. Moreover the framework is able to create a compiler which reads programs of the given grammar and produces models which conform to the generated metamodel. The generation of the metamodel can be adjusted by a predefined set of annotations which can be written directly into the grammar, so the generated model is more appropriate for whichever application.

Keywords: metamodels, grammars, programming languages, model transformation

1 Introduction

Model transformations are a key point in the ongoing research on model driven software engineering. Especially the ability to transform models from different modelling languages into each other is a crucial technology especially for the development and use of domain specific languages.

However the majority of today's programs was not created in a model-driven context but (more or less) directly written in a textual programming language. Since those legacy programming languages lack a proper metamodel, programs written in such programming languages can not be processed by model-driven software tools. This gap could be filled by automatically generated metamodels from grammars since most textual languages are defined by grammars. Once a metamodel for a

[1] Email: kunert@informatik.hu-berlin.de

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2008.04.034

textual language is defined, all programs written in this language can be treated as models (which are instances of the language's metamodel). The concrete transformation from programs to models is more complicated, but the main difficulty of the whole approach lies in the generation of a proper metamodel.

Various applications in the field of model-driven software development would benefit from the ability to treat grammar-based languages as metamodels and programs as models. For instance software reengineering and reconstruction would become significantly easier if the affected programs could be processed by visual modelling tools. The OMG (Object Management Group) defined the term *architecture driven modernization* (ADM [7]) for exactly this task. A similar but different application is the development and use of textual domain-specific languages, which would be the opposite approach to the one described in [6].

In this paper, we will present a framework which is able (i) to generate metamodels from grammars and (ii) to transform programs into models. Moreover, we present a qualitative characterization of the generated metamodels in order to facilitate further model transformation/refinement steps.

The rest of the paper is structured as follows. After this introduction we present some related works. In section 3, we define a measurement of quality which is needed to understand the proposed framework described in section 4. Section 5 describes our current implementation, while section 6 summarizes the paper.

2 Related Work

There are a couple of papers dealing with the connection between grammars and metamodels. A rather formal approach was taken in [1]. In this paper the authors define a relation between grammars and metamodels and describe a mechanism to convert instances from both concepts into each other. Unfortunately, the paper is restricted to the metamodel (M2) level. This implies that the transformation between concrete programs and models (instances of grammars and metamodels) is not handled. Moreover the generated metamodels are rather "flat" due to the fact that they are just different representations of the grammar rules (we will discuss this kind of "quality" of a metamodel in details in section 3).

The solution first described in [10] (and later in [2]) goes further. In this paper the author describes the generation of a metamodel for the ITU-T language SDL [3]. His approach was to generate the metamodel in two steps. A very simple metamodel was generated fully automatically from the grammar and then transformed in a number of manual steps until the metamodel had become a metamodel that was considered sufficient. However, this approach has two drawbacks: (i) these model transformations are not generic as they are only able to generate the metamodel of SDL and (ii) the model level (M1) is still not handled.

A relation between grammars and metamodels on metamodel *and* model level (M2 and M1) is discussed in [12]. The authors of the paper propose a framework which also works in two steps: model generation and model refinement. Moreover they propose the automatic generation of a model generator which produces models

```
program:          (vardefinition | assignment)∗
vardefinition:    type IDENT !SEMICOLON
type:             (INT | FLOAT)
assignment:       IDENT !BECOMES expression !SEMICOLON
expression:       (IDENT | NUMBER) (PLUS expression)?
```

Fig. 1. A sample grammar describing a very simple programming language (the exclamation marks will be explained in section 4.1 as they are used by our framework)

from programs. The proposed framework looks very similar to the one proposed by us but there are differences in the details. They also propose the automatic generation of simple metamodels which will be improved in later model transformations. However we have different opinions about when to solve which specific task and where to annotate the additional information needed to improve the metamodel. While they rely only on model transformation we start to improve our first metamodel before it is even generated. Moreover they propose to add the additional information into the metamodel created in the first step while we do not want to change intermediate models since they will be overwritten if we start the whole generation process again (e. g. if there is a slightly change in the underlying grammar). We propose to add all additional information into the grammar instead.

3 A characterization of metamodel quality

3.1 A measurement of quality

Before we start describing our proposed framework we want to introduce a measurement of quality of metamodels. This is necessary to understand various decisions made in the development of our framework.

As presented in [1], [2] and [12] it is easy to define generation rules, which produce a valid metamodel for a given grammar. Most of those ad hoc algorithms:

(i) generate classes for each grammar symbol (metasymbol and terminal)

(ii) introduce additional classes for occuring sequences, alternatives, optional parts and recurrences

(iii) connect all generated classes according to the grammar rules (by using aggregations, associations and/or generalizations)

In figure 1 a sample grammar for a simple language is shown. Figure 2 shows a metamodel which is created by an ad hoc algorithm.

Metamodels generated by such simple algorithms are not in principle bad or useless. It mainly depends on what the metamodels shall be used for.

Whenever you can define an application for your metamodel, you instantly get a measurement of quality. This measurement is defined rather pragmatically: the more appropriately the metamodel satisfies your needs, the higher quality it has. This implies that there is no global measurement of quality but many local ones.

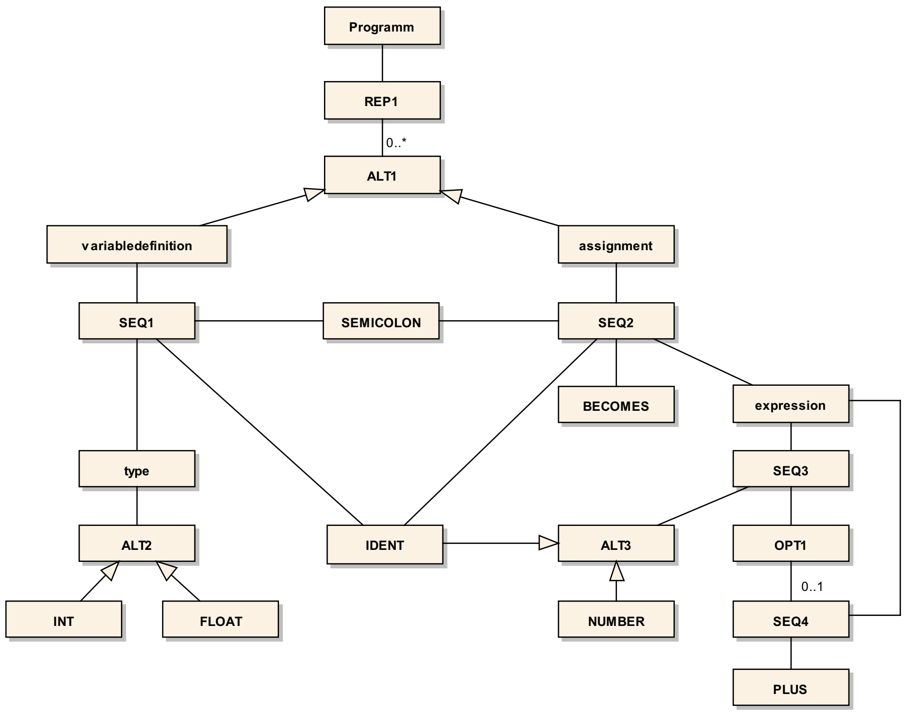

Fig. 2. Metamodel for the example grammar generated by an ad hoc algorithm

When we talk about higher or lower quality metamodels in this paper we consider the application mentioned in the introduction. We are especially interested in metamodels whose instances (i. e., models) can be easily transformed into other models which are instances of other metamodels.

3.2 A high quality metamodel

A metamodel suitable for our needs has to fulfill different requirements. First of all it has to be as abstract as possible (without losing any semantic detail of course).

This implies that it has to represent the semantics of the language and not the syntax. Therefore it is a good idea to start with an abstract grammar and not with a concrete one (as done in [10]). When no abstract grammar is given (as in most textual languages) it is useful to create one by stripping all terminals from the concrete grammar that are only needed for the concrete syntax (e. g. semicolons as statement separators are only needed in the concrete syntax and can therefore be deleted when generating the abstract grammar).

Moreover all constructs which only appear in the metamodel because of the concrete syntax of the grammar as defined by EBNF should be deleted. This means all helper classes for options, repetitions, sequences and alternatives.

Another concept from grammars no longer needed in metamodels are identifiers. Identifiers are only used in programs to reference other parts of the program, but in metamodelling we do not need this helper construct. Associations between referring and referred objects should be used instead. A by-product of this approach is that

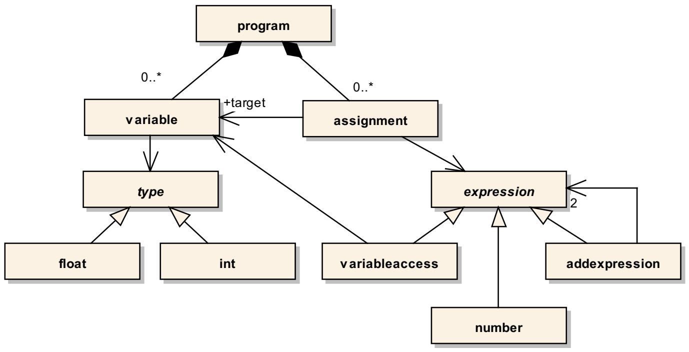

Fig. 3. A higher quality metamodel

the models become independent from a concrete identifier notation meaning that if we transform between models of different programming languages, the compatibility of identifiers in the concrete syntax does not have to be checked.

The last and most complicated requirement on a good metamodel for the applications mentioned in the introduction is the efficient use of abstract concepts. Abstract concepts are concepts which appear in different languages (e. g. the concept *namespace* appears as *namespace* in C++, as implication of *package* in Java and so on). It is very convenient to use abstract concepts since subsequent model transformations between different languages become a lot easier if you can simply rely on the mapping between the related abstract concepts of each language.

Figure 3 shows a metamodel which has a much higher quality according to the mentioned requirements than the metamodel of figure 2.

4 Description of the proposed framework

We propose a framework which generates metamodels and models from given grammars and programs, respectively. This generation is performed in two steps. The first step consists of the production of rather simple metamodels and conforming models using traditional compilers with a connection to a MOF repository. Since the generated models are low quality ones (as described in the previous section) there is a second step which transforms the low quality metamodel into a more appropriate one. Figure 4 gives an overview of the proposed toolchain.

4.1 The model generation part

The compiler part itself consists of two compilers called the *parent compiler* and the *child compiler*. The *parent compiler* reads a grammar written in EBNF and produces a metamodel by using a MOF repository. The EBNF grammar can contain some annotations influencing the metamodel generation. For instance you can mark all terminals which only belong to the concrete syntax but not to the abstract syntax (in our current implementation the exclamation mark is used as shown in figure 1), so they will not become an object in the generated metamodel.

The algorithm for the metamodel generation is a modified and corrected variant

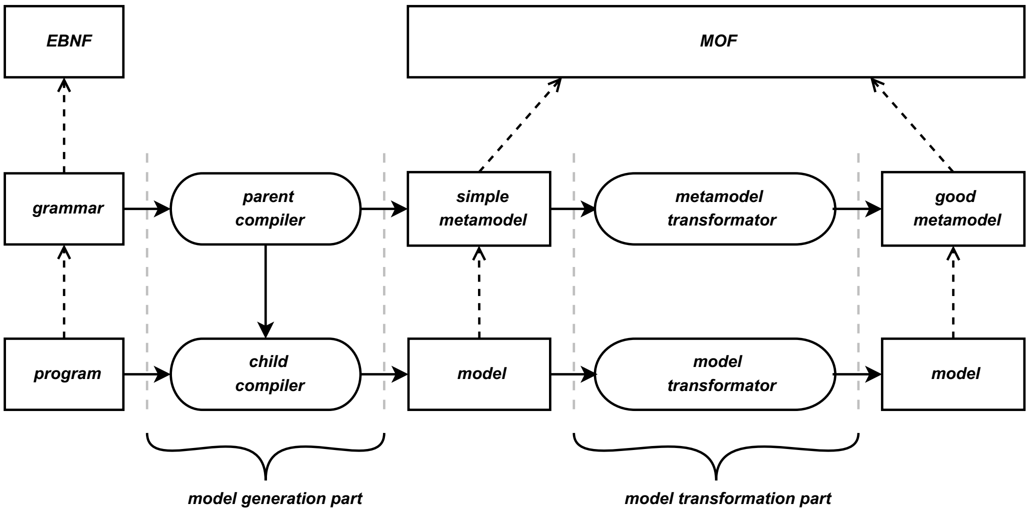

Fig. 4. Overview of the proposed framework

of [2]. It consists of three functions shown in figure 5 and 6 (written in pseudocode). The algorithm is a kind of improved ad hoc algorithm: classes are created for every symbol which is not marked for deletion. Then the classes are connected as defined by the grammar, but obviously unnecessary grammar-related constructs are not even created.

Figure 7 shows the metamodel which has been generated by the algorithm presented in figure 5. The difference between our algorithm and the ad hoc algorithms can be seen if figure 2 is used for comparison.

The *parent compiler* also generates the source code of the *child compiler* which is able to parse programs written in the language described by the given grammar. As a consequence a separate *child compiler* is generated for each programming language. Moreover the *child compiler* produces models conforming to the metamodel produced by the *parent compiler*.

4.2 The model transformation part

Once we have derived a simple metamodel by using the algorithm defined in the previous section, we can start improving it (to gain a metamodel with a higher quality for our purpose). Therefore we propose the introduction of additional annotations written in the grammar read by the *parent compiler*.

For instance identifiers in the metamodel can be deleted if we mark every definition and every use of an identifier with special annotations. Additional annotations can be used for the purpose of renaming classes in the metamodel.

We are still investigating how the introduction of abstract concepts can be expressed by grammar annotations which is by far the most complicate model transformation in our work.

The inclusion of annotations into the grammar file is convenient for the user since he has only to cope with one source file to control the whole model generation

```
function createMetamodel()
for each grammar symbol $X$
    if $X$ is not marked for deletion
      create a class named $X$
for each grammar rule $A \rightarrow B$
    if $B$ consists of only one symbol
      connect(A, B, "association")
    else if $B$ is an alternative ($B = B_1|B_2|\ldots|B_n$)
      for each $B_i$
        connect($A$, subexpression($B_i$), "generalization")
    else if $B$ is a sequence ($B = B_1B_2\ldots B_n$)
      for each $B_i$
        if $B_i$ is a repetition ($B_i = (B'_i)^+$)
          connect($A$, subexpression($B'_i$), "associationMult")
        else if $B_i$ is an option ($B_i = (B'_i)?$)
          connect($A$, subexpression($B'_i$), "associationOpt")
        else if $B_i$ is an optional repetition ($B_i = (B'_i)*$)
          connect($A$, subexpression($B'_i$), "associationOptMult")
        else
          connect($A$, subexpression($B_i$), "association")
    else if $B$ is a repetition ($B = (B')^*$)
      connect($A$, subexpression($B'$), "associationMult")
end function

function subexpression(expression $B$)
if $B$ consists of only one symbol
    return $B$
else if $B$ is an alternative ($B = B_1|B_2|\ldots|B_n$)
    create a class with a unique name $C$
    for each $B_i$
      connect($C$, subexpression($B_i$), "generalization")
    return $C$
else if $B$ is a sequence ($B = B_1B_2\ldots B_n$)
    create a class with a unique name $C$
    for each $B_i$
      if $B_i$ is a repetition ($B_i = (B'_i)^+$)
        connect($C$, subexpression($B'_i$), "associationMult")
      else if $B_i$ is an option ($B_i = (B'_i)?$)
        connect($C$, subexpression($B'_i$), "associationOpt")
      else if $B_i$ is an optional repetition ($B_i = (B'_i)*$)
        connect($C$, subexpression($B'_i$), "associationOptMult")
      else
        connect($C$, subexpression($B_i$), "association")
    return $C$
else if $B$ is a repetition ($B = (B')+$)
    create a class with a unique name $C$
    connect($C$, subexpression($B'$), "associationMult")
    return $C$
else if $B$ is an option ($B = (B')?$)
    create a class with a unique name $C$
    connect($C$, subexpression($B'$), "associationOpt")
    return $C$
else if $B$ is a optional repetition ($B = (B')^*$)
    create a class with a unique name $C$
    connect($C$, subexpression($B'$), "associationOptMult")
    return $C$
end function
```

Fig. 5. Our algorithm for the generation of the first (low quality) metamodel (Part 1)

```
function connect(expression $A$, expression $B$, connectionType $T$)
if $T$ is "association"
    create an association between $A$ and $B$
else if $T$ is "generalization"
    create a generalization between $A$ and $B$
else if $T$ is "associationMult"
    create a association between $A$ and $B$ with the multiplicity 1...n
else if $T$ is "associationOpt"
    create a association between $A$ and $B$ with the multiplicity 0...1
else if $T$ is "associationOptMult"
    create a association between $A$ and $B$ with the multiplicity 0...n
end function
```

Fig. 6. Our algorithm for the generation of the first (low quality) metamodel (Part 2)

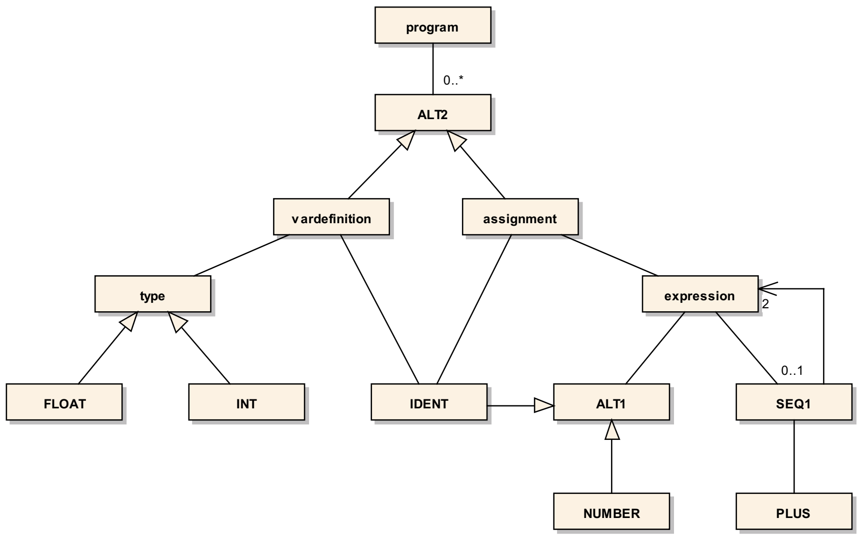

Fig. 7. Metamodel for the example grammar generated by our algorithm

process. Moreover later changes in the grammar can be done quite easily.

5 Implementation

We are currently working on a prototype of the proposed toolchain. We already implemented a working *parent compiler* producing simple metamodels and corresponding *child compilers*.

One of our implementation goals was to rely as much on standards and standardized tools as possible, so that our project results can be easily adapted by others.

The *parent compiler* is written with the help of the widely-used compiler generator ANTLR [8], but the source code is easily portable to any other LL(1)-parser generator (e. g. JavaCC [4]). The *child compiler* is also defined by ANTLR grammar specifications generated by the *parent compiler*.

The repository used by the parent and child compilers for model generation and later for model transformation is a MOF 2 repository called *A MOF 2.0 for Java* [11,9]. Since the only interface used to communicate with the repository is JMI [5], the repository can be exchanged with any other JMI-conforming MOF 2 repository without any further changes needed in our implementation.

6 Conclusion

The proposed framework is able to generate metamodels for every given grammar. The generated metamodels are not just other representations of the grammar, but metamodels which only contain the semantic information of the programming language and are therefore a good starting point for further model transformations.

Moreover our framework is able to automatically produce compilers which read programs written in the given languages and produce models according to the generated metamodels.

Once our implementation is finished we have a good base to migrate programs written in many textual languages into the field of (meta-)modelling. This implies that we can use all available model-based tools for software development, reengineering, modernization, etc. on programs written in legacy languages, which will make the mentioned applications much more understandable, easier and less error-prone.

References

[1] Alanen, M. and I. Porres, *A relation between context-free grammars and meta object facility metamodels*, Tucs technical report no 606, Turku Centre for Computer Science (2003).

[2] Fischer, J., M. Piefel and M. Scheidgen, *A metamodel for SDL-2000 in the context of metamodelling ULF*, in: D. Amyot and A. W. Williams, editors, *System Analysis and Modeling: 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1–4, 2004, Revised Selected Papers*, Lecture Notes in Computer Science **3319 / 2005** (2005), p. 208.

[3] ITU-T, "ITU-T Recommendation Z.100: Specification and Description Language (SDL)," International Telecommunication Union, 2002.

[4] java.net, "JavaCC – Java Compiler Compiler," Last checked: February 2006.
URL `http://javacc.dev.java.net/`

[5] JMI, "The Java Metadata Interface (JMI) Specification (Final Release)," Java Community Process, 2002, JSR-000040.

[6] Muller, P.-A. and M. Hassenforder, *HUTN as a bridge between modelware and grammarware*, in: *WISME Workshop, MODELS/UML 2005*, Montego Bay, Jamaica, 2005.

[7] Object Management Group, "Architecture Driven Modernization (ADM)," Last checked: February 2006.
URL `http://adm.omg.org`

[8] Parr, T., "ANTLR – Another tool for language recognition," Last checked: February 2006.
URL `http://www.antlr.org`

[9] Scheidgen, M., "A MOF 2.0 for Java," Last checked: February 2006.
URL `http://www.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava`

[10] Scheidgen, M., "Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000," Diploma thesis, Humboldt-Universität zu Berlin (2004).

[11] Scheidgen, M., *CMOF-model semantics and language mapping for MOF 2.0 implementation*, in: *Joint Meeting of the 4th Workshop on Model-Based Development of Computer Based Systems (MBD) and 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, 2006.

[12] Wimmer, M. and G. Kramler, *Bridging grammarware and modelware*, in: *MoDELS Satellite Events*, 2005, pp. 159–168.