How to represent Models, Languages and Transformations?

Martin Feilkas
Technische Universität München
feilkas@in.tum.de

Abstract

One of the main goals of domain-specific languages is to enable the developer to define completely new languages for special domains in order to get the advantages of programming on a higher level of abstraction. Nowadays languages are represented in different ways: by metamodels specified in some data modelling technique or by formal grammars. This position paper examines the influence of the representation of languages on language construction and transformation.

Introduction

In the last few years domain-specific languages (DSL) have been getting more and more attention in the software industry. DSLs could be a technique to develop software in shorter time and in better quality. DSLs promise to be a good solution to the problem of reuse not only on technical but also on architectural and design level. The usual way of handling this kind of reuse is the adoption of design or architecture patterns. DSLs can be seen as executable patterns. DSLs and generative techniques give the chance of defining the variability in specific software domains. Best practices, such as patterns, can be included as static parts in the generators and variable parts of a software system can be specified in some kind of model or language [GP]. Thus, DSLs present new perspectives on the development of software product lines.

But DSL development is still hard because domain and language development knowledge are required [WaH]. To make a DSL usable three tasks have to be carried out:

- Definition of an abstract syntax
  Most DSL-tools (also called language workbenches [LW]) allow the definition of the abstract syntax as a metamodel [MOF]. This metamodel is defined by a data modelling technique (the meta-metamodel) similar to class diagrams or ER-diagrams.
- Definition of a concrete syntax
  To make the language usable some concrete syntax has to be defined. Many language workbenches like the Microsoft DSL-Tools focus on graphical languages [MSDT, SWF]. For every language element there has to be a graphical icon that represents the abstract model element. Finally some kind of development environment needs to be provided. In the case of textual languages the syntax can be described by a grammar. A grammar describes both concrete and abstract syntax by specifying terminals, non-terminals and production rules.
- Definition of semantics
  Possibly the most important part of language specification is the formulation of semantics. An informal description of the language may be given in natural language by describing the domain itself. But the actual definition of these semantics is done by implementing the generator backend. Thus, the semantics of the DSL is defined by giving a translation (translational semantics) into some target language which already has some behaviour definition for its elements (operational semantics).

The generator backend is most often realized by one of the following three kinds of approaches [LOP]:

- Templates
  The most preferred approach to code-generation in language workbenches is the use of template techniques. As the name suggests code-files in the target language are the basis. Expressions of a macro language are inserted that specify the generator instructions. Often ordinary programming languages are used to specify the behaviour of the generator, e.g. C# in the Microsoft DSL-Tools [MSDT]. Other template languages like openArchitectureWare's functional Xpand language [oAW] specify a specific path through the model graph in each template by using recursion.

- Patterns
  This approach allows specifying a search pattern on the model graph. For every match a specific output is generated. This approach is used in BOTL [Botl1, Botl2] or ATL [ATL03].

- Graph-traversing (visitor approach)
  This kind of code generation approach specifies a predetermined iteration path over the syntax graph. For every node type, generation instructions are defined which are executed every time such a node is passed. This kind of generation approach is mainly used in classical compiler construction and textual languages.

Most language workbenches offer poor assistance for the specification of the generator back-ends. Ordinarily there are only little syntax-highlighting (only for the generator language but not for the target language) or code-completion features. The reason lies in the independency of the generator backend from the target language and the missing definition of the target language.

Today many languages are developed that are incomplete in the sense that manual coding is still needed to get an executable program. The adoption of DSL technologies is useful especially when the target code doesn't need to be touched after generation. Otherwise the developer using the DSL must still have full knowledge of the platform and the architecture of the generated code. In this case the benefits of the DSL's higher level of abstraction don't really take effect. In the early days of compiler construction generated machine code was also manually modified. This inconvenient practice was no longer necessary when the optimization techniques evolved in compiler construction. The same effect will probably take place when DSL techniques are further developed. But nowadays the reasons for manual coding in generated code are not performance issues but the difficulty to specify languages that are capable of expressing more than architectural and design decisions (like component or service structures). It would often be useful to be able to write logical or arithmetical expressions in a DSL. But it is cumbersome to specify this in a metamodel. Such common language constructs would be useful in many domains so the demand for reuse of language concepts arises. Manual modifications in generated code should be forbidden not only because of convenience reasons. It is a prejudice that generated code is less maintainable than hand written code. Manual interference may possibly destroy the architectural decisions specified in the DSL and its generator. Also, some typical technical round-tripping problems [RTE] could be avoided. For example, manually written code is lost when the generator needs to be run again due to changes of the model. Common solutions to this problem often lead to poor designs and bad program structures because of the inappropriate use of the target language's concepts (e.g. inheritance). This problem becomes obsolete if complete code could be generated out of DSL specifications.

In most language workbenches graphical languages are formulated as data models as the metamodelling technique of the workbench. These are usually simplified class diagrams or entity-relationship models (ER-models). In the vocabulary of the Meta Object Facility [MOF] this would be the meta-metamodel. Textual languages on the other hand are usually defined by their grammar, e.g. in Backus-Naur-Form (BNF).

The next section will compare these different representations of languages. Class diagrams can easily be transformed into ER-diagrams. Due to that we will not distinguish between these data modelling techniques anymore and only talk about ER-modelling and relational models in the next sections. After that we will describe the effects of a uniform meta-metamodel on the generation and transformation techniques. At last we want to address the composition of languages out of language components.

Data Modelling vs. Grammars

As mentioned above the big difference between classical compiler construction and language workbenches is the formulation of the metamodel. Compilers use formal languages whereas generators use ER-models. The problems compiler construction is facing arise because of the linearity of text. It is difficult to encode and decode information into a linear representation (parsing). Recognizing and reconstructing the information which is encoded into text makes it necessary for every compiler to solve the word problem to decide whether a given program is syntactically correct and in order to reconstruct what the programmer had in mind when writing the program.

An interesting question concerns which is the better or more expressive way of formulating metamodels. We will examine this topic using a small example. Figure 1 shows a simplified part of an abstract syntax tree of an ordinary imperative language.

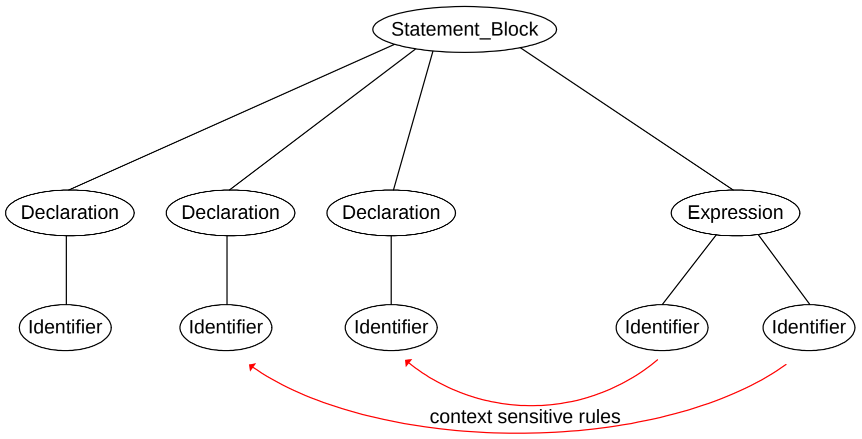

Figure 1: A simplified abstract syntax tree

This simple example shows the definition and the use of identifiers. An ER-schema whose stored data represents the same information as the syntax tree (without the context sensitive rules) would look like this:

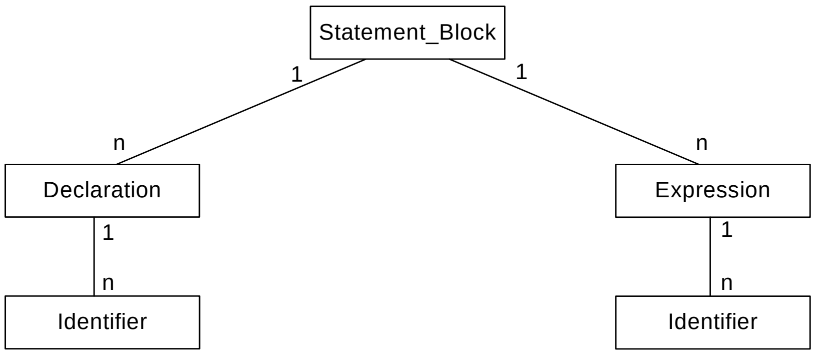

Figure 2: An ER-model as a metamodel for the syntax tree in Figure 1

But a better way of expressing all the information needed for the concept of usage and declaration of identifiers would be the following:

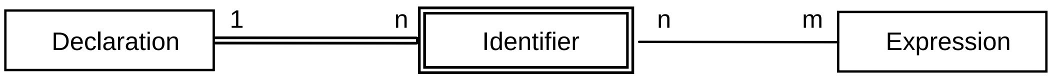

Figure 3: An ER-diagram representing the declaration and usage of identifiers

This shows a simple example of the weaknesses of compiler construction: It is impossible to express all the information needed in context free grammars. In almost every (non-trivial) language there are context sensitive rules that must be integrated and checked by manually coding the compiler. A similar example would be the definition of interfaces in Java or C#. If a class implements an interface it has to implement all the methods declared in this interface. This rule is also context-sensitive and cannot be expressed in a BNF notation of these programming languages. In every case where information (identifiers or method-signatures) is specified more than once in programs a relational structure can be found that eliminates this redundancies and expresses both, the information within the syntax tree and the context-sensitive information. This is possible because a database schema is not only capable of storing trees but also universal graph structures (with typed nodes).

The presence of context-sensitive rules of course lowers the maintainability of the compiler. Using context sensitive grammars is not possible either, because they are not easy to handle and the word problem (check if a given word is part of a language) cannot be solved efficiently. The examples above already show the assumption that ER-models are more expressive than context free grammars. But the expressiveness of ER-models is limited, too. In the second example (interface implementation) there could also be the restriction that the interface-methods implemented by the class must be declared as public. This exceeds the expressive power of ER-modelling and constraints formulated in predicate logic would be needed.

Our goal is to define a way to translate context-free grammars into ER-schemata and optimize them towards the context-sensitive rules. By using normal form theory we will try to find a way to store programs without redundancy. More research is needed to formalize this topic. Further work will discuss this in more detail. Now we want to have a look at the advantages that could possibly be gained by using a relational representation of a language.

Benefits of relational metamodels

Keeping these advantages of data-modelling compared to grammar-based definition of languages in mind, the question arises if this technique could also be used in ordinary programming languages. The formulation of a programming language as an ER-schema and the storage of programs in a relational database would demand an extra definition of concrete syntax in a textual or graphical way.

Formulating languages as relational data schemata can make the use of a parser unnecessary because the programs would directly be stored as abstract syntax graphs and the construction of the program could be done syntax-driven. If a graphical program representation is preferred, the operations of dragging and dropping language elements onto the drawing board have to take care that either no incorrect models can be produced or at least that no incorrect model can be stored or executed by the generator backend. In the case of a textual representation of the programs ordinary parsing-techniques may be used before storing the abstract syntax graph or structured editors could be applied like it is done in the Meta Programming System [LOP]. Programming in this kind of relationally represented language would at last be a sequence of insert, update and delete operations whereas DSL composition would be done using add, alter and drop table operations.

Another advantage of storing programs in a relational format is much easier refactoring. For example, in a conventional programming language, a simple renaming of a method declaration would cause the programmer to identify and consistently change all of the usages of this method. This phenomenon is comparable to update anomalies known in database theory. In an ER-schema of a programming language, a method call could be realized by references (numeric primary key) without storing the method name twice [Sim06, Sim96].

Versioning systems (like CVS, Subversion) work on simple pattern matching practices and are unable to identify the reason of conflicts. With ER-based programming languages merge conflicts could be solved much more easily because the versioning system has all the details necessary to determine the exact reason for the conflict. Not only differences in the text lines can be shown, but detailed information on what the differences really rely on (e.g. a new method is declared or an old one has been moved). The developer could be informed of conflicts in a much more detailed way and do a kind of “semantic merge” [LW, Sim06, Sim96]. It is no longer possible to destroy the syntactic correctness of the code base by doing merging operations. Even scenarios are imaginable where developers work synchronously on the same copy of code stored in a central database. For every statement the author could be stored. Two developers would immediately notice when working on the same piece of code. This could avoid merge conflicts totally.

Data modelling is much easier than dealing with grammars. Software engineers are usually experienced at building data models. One of the main ideas behind DSLs is to give the programmer the ability to modify his own tool, the programming language. Compiler construction know-how is not very common in average software companies but if the modification of a language could be done via simply changing a relational schema this idea would get better acceptance. Some simple modification could be the implementation of company specific coding conventions by restricting the programming language used.

Model Transformation

After this reasoning about the way of defining metamodels, these observations should now be examined towards their influence on the generator backends and model transformation capabilities. DSL-workbenches generate just plain unstructured text, and it is not ensured, that the generator output really fits the grammar of the target language. The used generation techniques do not respect the target language syntax (respectively its metamodel). A formal mapping between the source and target language elements is very important because it specifies the semantics of a newly developed DSL. If not every word in the new language can be translated into an equivalent and syntactical correct target language word, how is the semantic of these words defined? In our opinion the semantics of DSLs must be specified as translational semantics. This leads to the need for syntax respecting code generation. By specifying a translation to a target language that has operational semantics a newly developed language implicitly gets a formal definition of its own semantics.

To ensure the syntactical correctness of the output of a transformation it would be necessary to define a separate transformation language for every pair of source and target language. Such a transformation language would consist of the syntax definitions of the source and the target language and some language elements needed to be able to define the mapping. The synthesis of such a transformation language is always the same process and could therefore be automated.

Usually people distinguish between model-to-code and model-to-model transformations. One of the difficulties in syntax-respecting model-to-code transformations relies on the different representations of models and code. Models are usually defined relationally and code by its underlying grammar. Actually four kinds of transformations should be distinguished:

1. ER-defined language to Grammar defined language
2. ER-defined language to ER-defined language
3. Grammar defined language to Grammar defined language
4. Grammar defined language to ER-defined language

If a language workbench would have a (relational) representation of a target language (e.g. a 3rd generation programming language) then the task of model-to-code generation could be treated equally to model-to-model transformations and techniques like ATL [ATL03] or QVT [QVT] could be applied. Several other concepts for model-to-model transformations have been submitted to the QVT request for proposals of the OMG [QVTr].

But these model-to-model transformation techniques have a different methodology than template based code generation. It is questionable if these would really fit the needs because most examples of model-to-model transformations just show transformations between models that have almost the same level of abstraction. Code generation on the other hand often has to deal with a huge gap between the levels of abstraction of the source and target language.

```
Component c → insert Java-Class(Name = c.Name, Visibility = public) jc {

   insert Constructor(Name = jc.Name);

   Port (Type == "sender")  p    → insert Method(Name = c.Name+"_"+p.Name){…};
   Port (Type == "receiver") p   → insert Method(Name = c.Name+"_"+p.Name){…},
                                   insert Method(Name = "CS"+c.Name+"_"+p.Name){…},
                                   insert Attribute(Name = "isCalled_"+p.Name, Type = boolean);

   DataElement d → GenAttr;
}

GenAttr : DataElement d → insert Attribute(Name = d.Name, Type = string);
```

Figure 4: Example of a fictitious transformation language

The example for a transformation language illustrated in Figure 4 can be read equally to ordinary template approaches and is inspired by the Xpand language used in openArchitectureWare [oAW]. The left hand side of the transformation rules can be thought of as model elements over which is iterated in a for-each-loop and the right hand side as the body of the loop which ordinarily generates the textual output. The right hand side specifies the abstract syntax elements that should be inserted into the target model when the rule is executed.

In contrast to approaches like ATL [ATL03], this kind of transformation is traversal based rather than pattern based. The transformation specifies a spanning-tree in the syntax graph of both the source and the target language, which is constructed by the right hand side of the transformation rules. So the transformation can be executed as a depth-first search over the abstract syntax graph of the source language word and at every node parts of the abstract syntax graph of the target language word are constructed. A real complex transformation may need several of these passes. An important thing about the specification of the transformation is that it is done in a declarative manner. Unlike text generation the transformation doesn't create a concrete syntax representation of the resulting target language word but its abstract syntax graph.

This transformation language is composed out of the source and target language and elements for defining the transformations. It is very context-sensitive because the transformation rules have to take care of the source and target language syntax and mapping constraints. But if it can be shown that a word is in the transformation language, this word is a complete definition of translational semantics for the source language.

Especially in cases where the source and target languages have a huge difference in their level of abstraction the transformation language in this example is not very useful, because many target elements have to be generated from single source elements and the transformation gets very hard to read. Ordinary templates have the advantage that the concrete syntax of the target language gives a good impression of what generator output is to be expected [PTM]. The example of the textual transformation language above is just meant to show in which manner transformations should be expressed. Of course the transformation language itself could (just as every other language) be represented as an ER-schema and a good GUI-concept could ease the task of specifying transformations by offering only the appropriate elements like code completion does in modern IDEs.

The definition of the semantics of the transformation language itself can be defined in a bootstrapping way by formulating a transformation from the transformation language to a programming language (with operational semantics) in the transformation language.

Language Composition

One of the big goals of this approach to code generation/model transformation is to make it possible to identify the dependencies between target elements and source elements to identify free elements. In the example the DataElement that corresponds to the attribute doesn't need information (fields) of the component in whose context it is generated. You can say that in this simple example the DataElement concept is independent of the component concept.

This could be a first conceptual step toward a composition of languages in a building blocks approach. By introducing new blocks of an existing language (e.g. arithmetic expressions) into a new language, all of the transformation rules that do not depend on other elements of the existing language can be taken over into the generator of the new language. So this is an approach to semantic conserving DSL-composition. Through similar concepts the development of complete DSLs can be done with less effort.

Conclusion and further work

In this position paper we have shown a new perspective on the differences between models and code. The main difference lies in the representation of their metamodels by relations or grammars. After that we explained what advantages could be expected, if ordinary programming languages are formulated as relational models. Translations between relationally represented languages have been proposed and discussed towards the goal of developing transformations that verify the syntactical correctness of the target words. This kind of transformation could also be an exact definition of the (translational) semantics of a newly developed language.

In the future we will work on an implementation of a relational programming IDE with a transformation language like the one presented. It has to be formally shown where relational modelling is situated in the Chomsky hierarchy.

References:

[ATL03] J. Bézivin, G. Dupé, F. Jouault, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In the online proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of the MDA, www.softmetaware.com/oopsla2003/mda-workshop.html.

[Botl1] P. Braun, F. Marschall: Transforming object oriented models with BOTL, Electronic Notes in Theoretical Computer Science 72, 2003.

[Botl2] F. Marschall, P. Braun: Model Transformations for the MDA with BOTL, Univeristy of Twente, 2003.

[GP] K. Czarnecki, U. Eisenecker: Generative Programming, Addison Wesley, 2000.

[LOP] S. Dmitriev. Language oriented programming: The next programming paradigm. Onboard Magazine, www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html, November 04.

[LW] M. Fowler: Language Workbenches: The Killer-App for Domain Specific Languages?, www.martinfowler.com/articles/languageWorkbench.html, Jun 05.

[ML] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood: Transformation: The missing link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, Proc. Graph Transformation - First International Conference, ICGT 2002.

[MOF] Object Management Group: Meta-Object Facility (MOF™) Version 2.0, www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF.

[MSDT] Microsoft DSL-Tools, http://msdn.microsoft.com/vstudio/DSLTools, August 06.

[MTA] K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches. In Proceedings OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[oAW] Open ArchitectureWare, Generator Framework, www.openarchitectureware.org.

[PTM] J. van Wijngaarden, E. Visser: Program Transformation Mechanics: A classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems, May 2003.

[QVT] Object Management Group: MOF QVT Final Adopted Specification, July 7, 2006, www.omg.org/docs/ptc/05-11-01.pdf.

[QVTr] Object Management Group: 2.0 Query / Views / Transformations RFP, April 24, 2002, www.omg.org/docs/ad/02-04-10.pdf.

[RTE] S. Sendall and J. Küster: Taming Model Round-Trip Engineering. In Proceedings of Workshop 'Best Practices for Model-Driven Software Development', Vancouver, Canada, October 2004.

[Sim06] Ch. Simonyi, M. Christerson, S. Clifford: Intentional Software. Proceedings of OOPSLA'06, 2006.

[Sim96] Ch. Simonyi, Intentional Programming - Innovation in the Legacy Age. Presented at IFIP WG 2.1 meeting, June 4, 1996.

[SWF] J. Greenfield et al.: Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. John Wiley & Sons, 04.

[WaH] M. Mernik, J. Heering, A. M. Sloane: When and How to Develop Domain-Specific Languages, ACM Computing Surveys, Vol. 37, No. 5, December 2006.