Challenges for advanced domain-specific modeling frameworks

István Ráth[*], Dániel Varró[†]
Department of Measurement and Information Systems
Budapest University of Technology and Economics[‡]

Abstract

In this paper, we identify main challenges for domain specific modeling frameworks. **Arbitrary abstract to concrete syntax mapping** enables the toolsmith to create visual representation to complex logical models with the possibility to determine what detail to show and what to hide in a flexible way. **Interactive model simulation** based on a model transformation architecture allows for the design time analysis and precise fine tuning of behavioral aspects.

1 Introduction

The main goal of language engineering is to integrate the following aspects into a coherent framework: (i) *abstract syntax*, which specifies how logical model elements are stored and represented in the modeling environment; (ii) *concrete syntax*, which specifies the graphical representation of models; (iii) *well-formedness rules*, to describe static and language-specific constraints; (iv) *dynamic (operational) semantics* which describe the behavioral attributes of the system-under-design; and (v) *transformations* which provide formal and declarative support for creating generative bridges between various modeling domains.

Many state-of-the-art tools, such as MetaEdit [6], Microsoft's DSL Tools [7], openArchitectureWare [2] or the upcoming Graphical Modeling Framework [9] solve the traditional problems of custom modeling (the first three aspects) in an elegant fashion: constructing language metamodels is easy, and deriving a usable graphical representation is either automatic, or it is facilitated using some additional models. Additionally, language-specific constraints can be declared and evaluated using some technique (e.g. OCL). Thus, minimal, or no manual coding is required for building a (basic) domain-specific modeling tool. This is a key factor in speeding up the development process, and therefore one of the most important research goals of our research is to provide fully declarative support (i.e. model-based, without requiring the developer to write code manually) to all stages of DSML development.

However, there are multiple key areas where existing DSM frameworks typically fall short. The problem of **abstract to concrete syntax mapping** arises from the fact that abstract syntax models tend to be too complicated and difficult to handle for humans. This is especially the case in model transformation environments where the optimal metamodel configuration for model transformation can radically differ from that of a visual designer. Therefore, in recent initiatives, such as Eclipse's General Modeling Framework [9], the modeling and diagram layers are conceptually separated.

Interactive model simulation is another important language engineering aspect which has been lacking support in traditional modeling tools. There have been attempts at providing some (limited) support for it based on common patterns such as state machines (e.g. in MetaEdit+ [6]), however a general implementation, integrated into a modeling environment is still to come.

The ViatraDSM Framework The development of the ViatraDSM Framework is an ongoing research activity at the Department of Measurement and Information Systems of the Budapest University of Technology and Economics. With ViatraDSM, our main goal was to push the limits of tool integration further, in order to ease and speed up the construction of rich domain-specific visual languages.

[*] istvan.rath@gmail.com
[†] varro@mit.bme.hu
[‡] This work was partially supported by the RESIST and SENSORIA European projects.

The ViatraDSM framework is tightly integrated with the VIATRA2 [1] engine, and relies on its modelspace containment and model transformation facilities to provide support for the evaluation of *well-formedness constraints*, *model simulation* and *interdomain transformations*.

In the following two sections, we provide a detailed summary of initial research challenges and results concerning two main topics introduced above: abstract to concrete syntax mapping and interactive model simulation.

2 Arbitrary abstract syntax to concrete syntax mapping

In constructing domain-specific visual languages, the clear separation of the logical modeling layer and the visualization layer is important. Abstract syntax models tend to be quite complicated, especially if they are tailored to the needs of the code generator, rather than the user. Therefore, the visualization layer needs to be independent, because that way the language engineer is free to construct diagrams that are easy to handle and do not confuse domain experts.

The developers of the Graphical Modeling Framework [9] (GMF) have also identified the importance of this clear distinction, which is apparent in GMF's multi-layered approach:

1. **domain (language) metamodel**, which defines the concepts of the language (abstract syntax);
2. **diagram metamodel**, which defines a diagram metamodel, including the graphical appearance of model elements (concrete syntax);
3. **mapping definition**, to create a bridge between the domain metamodel and its visual representation;
4. **tooling definition**, which defines how the user can edit models.

It is important to note the difference between the *model level* and the *metamodel level* separation of logical models and diagrams.

On the **model level**, most tools separate the two layers by giving the user (almost) complete control over what parts of the logical model are visualized. This means, that the user either creates logical models by editing the model on a diagram, or visualizes existing elements by dragging them onto a diagram, for instance. However, diagrams strictly follow the structure of the logical model, i.e. a single logical entity is always represented by a single graphical entity. This makes the graphical editing easier to implement, because the graphical representation can be directly mapped onto the logical domain.

The **metamodel level** separation, enables the language engineer to define visualization independently. Logical and diagram metamodels are constructed separately. This means that the GUI-driven editing only affects the diagram models directly, and the required changes to the logical model are generated by a mapping interface.

The mapping between the logical and diagram models can be approached in two ways: either by completely separating the two layers and implementing **bidirectional synchronization** between them (as promoted e.g. by the QVT standard). This is the broadest conceptualization of abstract syntax-concrete syntax separation, however it is difficult to implement due to its generality.

The other approach, **bidirectional mapping** makes use of the fact that the user can only apply a limited and well-defined set of modifications to the diagram through the graphical interface. Therefore, this approach directly maps the editing actions of the user to the logical model (see Fig. 1).

This mapping interface is also responsible for reflecting the changes of the logical models in diagrams. As the user is editing the models visually, the user's editing actions are transformed into commands that affect the logical model. The logical model, in turn, notifies the visualization layer that it has become "dirty". The mapper interface maps these notification objects to valid diagram-specific commands which eventually change the appearance of diagrams.

Mapping in textual languages The problem of mapping between abstract and concrete syntaxes of a visual language is similar to the problem in the domain of textual languages (as discussed in [5, 10]). The important difference is that in textual languages a separate "visualisation" modeling layer between the abstract syntax representation (e.g. an Abstract Syntax Tree) and the visual source code representation is absent. In visual languages, diagrams display a "projection", or aspect of the logical modelspace; its equivalent in textual languages would be an aspect-oriented editor which would enable the developer to only see

certain aspects of the source code.

The specification of the bidirectional mapping mechanism should ideally be fully declarative, in order to avoid the necessity of writing complicated and critically important code. GMF solves this problem by employing an EMF-based *mapping definition*, which plays a crucial role in the code generation process. This mapping definition, however, has inherent limitations. For example, GMF only allows the mapping of EClasses to nodes, and EReferences to edges. That is, it imposes a restriction on how diagram definitions can be assigned to logical models.

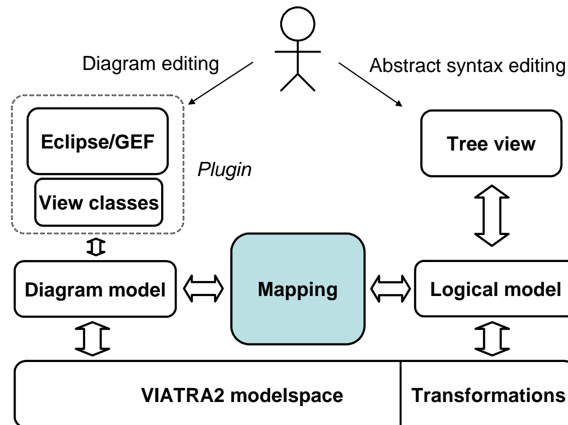

Figure 1: Bidirectional mapping interface

Although GMF allows for a partial representation of the logical model (meaning that not every logical feature has a graphical counterpart), it is rather limited in providing support for more advanced mappings. For instance, *attributes* or *properties* are frequently used in diagrams for the expression of the amount of model subelements assigned to a container. In a Petri net editor, for example, *place* nodes can contain *token* nodes. It is common to visualize the amount of tokens assigned to a place by displaying a single integer attribute as a decorator of the place's graphical representation. With the GMF approach, it's not possible to establish such a logical link between the amount of tokens assigned to the place, and its attribute.

Our goals ViatraDSM's goal is to overcome this restriction by using the pattern matching and graph transformation capabilities of the underlying VIATRA2 [1] engine. The implementation behind the mapping interface on Fig. 1 can be constructed in three ways:
(i) by using **manually written code** (utilizing a well-defined application programming interface);
(ii) by **code generation** based on a declarative specification;
(iii) based on an **interpreter** approach, which facilitates the translation of editing requests to commands and model notification objects to diagram modifications based on a "program" stored in the model space.

Currently, since prototypes based on the first approach are already implemented, active research is being conducted towards the interpreter-based approach.

3 Interactive model simulation

Design-time interactive *model simulation* based on dynamic (executable) semantics is another important language engineering aspect, which is not present in existing tools. The reason behind this is that it is difficult to implement in a way that is general enough to support a wide range of possible target domains. In the case of GMF [9], for instance, model simulation and transformation is out of scope, since the modeling infrastructure is entirely based on another Eclipse project, the Eclipse Modeling Framework [8]. Today, it is not straightforward how GMF/EMF can interoperate with model transformation techniques.

Since the ViatraDSM framework is built on the VIATRA2 model transformation infrastructure, interactive model simulation can be implemented more easily. Ongoing research focuses on the following areas:

- Guided simulation using a **step-based execution**. This can be implemented with parametrized, single-step transformation rules. In that case, the toolsmith constructs a model transformation which executes a single transformation step. The framework runs these transformation steps, with support for user interaction at non-deterministic choice points.

- **Debug-style simulation** with support for more precise control over the transformation process. This can be regarded as an evolutionary continuation of the step-based execution approach.

  The model transformation capabilities of the VIATRA2 system are based on a graph transformation (GT) [4] approach for the description of model transformation rules, and an abstract state machine

(ASM) [3] specification for the control flow. This approach provides an interface to the pattern matcher and the GTASM interpreter, where a user can place breakpoints into the transformation program, and inspect and change (pattern) variables etc.

From the point of view of the ViatraDSM framework, the most challenging aspect of this approach is to provide a domain-specific interface to the debugger.

- General **trace representation** to preserve the different states of the model during simulation. Tracing is an important aspect of model simulation, because it provides useful feedback on simulation runs. With a trace, the various states of the system and the non-deterministic choices can be regenerated at a later time.

  This research topic is strongly connected to those of debug-style simulation, because traces should ideally have a domain-specific interface as well.

4 Open Issues

To sum up, current research is active in the following areas:

1. Arbitrary model-to-diagram and diagram-to-model mapping based on a declarative specification, to facilitate the metamodel level separation of diagrams and logical models, and provide a flexible and transparent bidirectional mapping between them.

2. Design time interactive model simulation and constraint evaluation using transparent model transformation techniques, with support for the generation of traces. A domain-specific interface to the generic transformation engine is in the planning stage.

3. Multi-domain modeling using transparent model transformation techniques, to allow for multiple domain-specific views of the same logical model instances. A typical shortcoming of current DSM tools is **domain integration**. The end result of the custom language building process in all of todays tools is a standalone visual language. However, it is often necessary to use multiple domains simultaneously during system design, especially for large and complex systems. A typical example of this is the application of Performance or Reliability profiles to structural models in order to evaluate the system under design at the earliest possible stage from various perspectives.

References

[1] *VIATRA2 Framework*. An Eclipse GMT Subproject (`http://www.eclipse.org/gmt/`).

[2] Bernd Kolb, Markus Völter. openArchitectureWare and Eclipse. `http://architecturware.sourceforge.net/data/oawEclipse.pdf`.

[3] E. Börger and R. Särk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[4] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[5] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[6] Metacase. MetaEdit+. `http://www.metacase.com/mep/`.

[7] Microsoft. DSL Tools. `http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx`.

[8] The Eclipse Project. Eclipse Modeling Framework. `http://www.eclipse.org/emf`.

[9] The Eclipse Project. Graphical Modeling Framework. `http://www.eclipse.org/gmf`.

[10] D. S. Wile. Abstract syntax from concrete syntax. *Proceedings of the 19th International Conference on Software Engineering*, 1997.