Abstracting Complex Languages through Transformation and Composition*

Jendrik Johannes[1], Steffen Zschaler[2], Miguel A. Fernández[3], Antonio Castillo[3], Dimitrios S. Kolovos[4], and Richard F. Paige[4]

[1] Technische Universität Dresden, jendrik.johannes@tu-dresden.de
[2] Computing Department, Lancaster University, szschaler@acm.org
[3] Telefónica Research & Development, mafg@tid.es,acastillo@polar.es
[4] Department of Computer Science, University of York, dkolovos,paige@cs.york.ac.uk

Abstract. Domain-specific languages (DSLs) can simplify the development of complex software systems by providing domain-specific abstractions. However, the complexity of some domains has led to a number of DSLs that are themselves complex, limiting the original benefits of using DSLs. We show how to develop DSLs as abstractions of other DSLs by transfering translational approaches for textual DSLs into the domain of modelling languages. We argue that existing model transformation languages are at too low a level of abstraction for succinctly expressing transformations between abstract and concrete DSLs. Patterns identified in such model transformations can be used to raise the level of abstraction. We show how we can allow part of the transformation to be expressed using the concrete syntax of the concrete DSL.

1 Introduction

Domain-specific languages (DSLs) [1] are used to reduce the complexity arising when developing software systems using general-purpose languages (GPLs). A DSL contains a relatively small number of constructs that are immediately identifiable to domain experts and allow modellers to construct concise models capturing the design of the system at an appropriate level of abstraction. While DSLs typically start off with a small number of constructs, they tend to grow over time: as they are used, new concepts, features and relationships are identified and are subsequently added to the DSL—making it more flexible within a wider domain. This flexibility introduces accidental complexity as modellers need to make decisions about using each feature. This can eventually compromise the very aims for which the DSL was built: domain focus and conciseness. However, the additional concepts and features have been added for specific purposes and cannot be simply dismissed; the complexity of the DSL is intentional. One example of such a complex DSL is the Common Information Model (CIM)

* This research has been co-funded by the European Commission within the FP6 projects MODELPLEX (contract no. 34081) and AMPLE (contract no. 33710)

[2] DSL for network configuration. While CIM is a DSL, its size in terms of the number of concepts and features it contains has progressively become comparable to that of a GPL such as the UML. Many of the internal details of devices in a network configuration, however, are quite irrelevant when we try to model and understand the configuration as a whole. Still, these details are very much relevant when the configuration is to be implemented or manipulated.

This paper is about how we can efficiently develop layers of DSLs; that is, new DSLs that provide abstractions of the concepts in existing DSLs. Such abstractions also may help to obtain models with desirable properties, e.g., models that can be more easily navigated for transformation purposes. The abstractions can also support ensuring the correctness of the models by construction instead of relying solely on post-construction verification, by allowing only particular combinations of model elements to be used.

The main contribution of this paper is the presentation of a generic translational approach for abstract DSLs, based on the identification of patterns in model transformations. The approach works for all situations where each concept in the abstract DSL can be translated into a partial model in a concrete DSL. Complete models in an abstract DSL are then translated into compositions of partial models in the concrete DSL. The approach has been implemented prototypically using existing model transformation and composition technologies. Due to space restrictions, we cannot give more than a general overview. Readers interested in a more detailed discussion are referred to [3].

An obvious solution to the problem of building layers of DSLs is to design a more *abstract* DSL using a standard modelling framework (e.g. EMF) and then to use a model-to-model (M2M) transformation language such as QVT, ATL or ETL to transform models expressed in the *abstract* DSL, into models that conform to the *concrete* DSL. The main advantage of this approach is that it is based on robust and well-understood technologies. Nevertheless, having written several such abstract DSLs and transformations we have also identified several shortcomings. First, the produced transformations are very much alike and demonstrate several recurring patterns which need to be implemented from scratch every time. Moreover, as single elements in models expressed in the *abstract* DSL typically correspond to fragments consisting of several elements in the target models that conform to the *concrete* DSL, constructing such fragments needs to be done programmatically in the context of the M2M transformation—which we have found to be counter-intuitive and error-prone.

If creating an abstract DSL from a more concrete one was a one-off, building new tool support for automating it would most likely be unreasonable. From our experience in providing tool-support for DSLs for industrial partners in the ModelPlex EU project, this appears to be a recurring pattern. To address the aforementioned shortcomings in a systematic way, a mechanism is needed which allows developers to abstract from the commonalities of these concrete-to-abstract DSL mappings and specify the mapping logic in a high level declarative formalism that provides first-class support for recurring patterns.

2 Language Mapping Patterns

We have identified the following recurring patterns in the relationship between abstract DSL model elements and concrete DSL model elements:

1. *Element Mapping.* This pattern embodies the fundamental form of abstraction in our scenario: the representation of a recurrent configuration of concrete DSL model elements by a single model element in the abstract DSL.
2. *Element Mapping with Variability.* This pattern maps an abstract DSL model element to a network of model elements of the concrete DSL. The model elements in the network and their connections are selected based on the value(s) of one or more attributes of the abstract DSL model element.
3. *Attribute Mapping.* This pattern maps the value of an attribute of an abstract DSL model element to the value(s) of one or more attributes of concrete DSL model element(s). This mapping pattern is essential because, unlike the two patterns discussed above, it allows concrete data values to be passed from the abstract DSL model into the concrete DSL model.
4. *Link Mapping.* This pattern maps a link between two abstract DSL model elements to one or more links between concrete DSL model elements. This pattern is essential to translate relationships between elements in an abstract DSL model into relationships in a concrete DSL model.

To make transformations easier to understand and write, it would be useful to make explicit the use of each mapping pattern. That is, rather than manually writing the complete transformation, one could, for example, annotate the abstract DSL metamodel with appropriate mapping patterns and generate the transformation from these annotations in an automated manner. Generating the transformation has the added benefit that each pattern can be implemented consistently wherever it is instantiated.

By modelling the concrete-DSML configurations as separate model fragments and referencing these fragments from the pattern annotations, we can further improve our specifications. This avoids cluttered specifications, allows the use of concrete-DSML editors for defining large parts of the transformation, and can remove scattering and tangling from the transformation specification [3].

3 Implementation

Here, we present a prototypical implementation of our approach, based on the Reuseware Composition Framework [4] as well as the Epsilon Transformation Language (ETL) [5] and the Epsilon Generation Language (EGL) [6]. Figure 1 gives an overview of the prototype and the process of using it. Most of the steps presented are automatic, artefacts that need to be provided to the prototype have been highlighted in grey in the figure. There are two phases to using the prototype: The first phase (named 'meta level' in Fig. 1) comprises the design of the abstract DSL, while the second phase (named 'model level' in Fig. 1) starts when the abstract DSL is used.

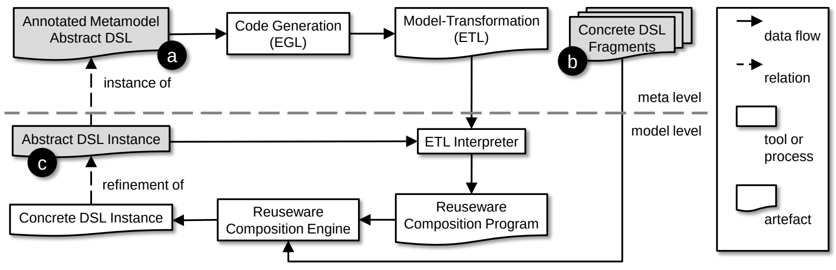

Fig. 1. Overview of the architecture of our prototypical implementation

To define a new abstract DSL, language designers need to provide two artefacts: 1) a metamodel of the abstract DSL (labelled ‘a’ in Fig. 1), annotated to define the transformation to the concrete DSL, and 2) a set of partial models (labelled ‘b’ in Fig. 1) that will be mapped to by this model transformation. Partial models are represented in our prototype through the notion of ‘Model Fragments’ defined in Reuseware. Among other things, developers can assign a *Unique Fragment Identifier (UFI)* to each model fragment. Reuseware then provides an API to obtain a model fragment by its UFI. These UFIs can be used in the metamodel annotations to uniquely refer to a fragment to map to.

From an annotated metamodel, our prototype then generates a model transformation program in ETL. The code generator that creates this model transformation is written in EGL and contains the definitions of the four patterns identified in Sect. 2. The generated ETL model transformation expects an instance of the abstract DSL metamodel and transforms this into a composition of the appropriate model fragments.

Reuseware provides a so-called *composition interface* for each fragment. This interface contains two types of named points: *reference points* and *variation points.* The former allow to extract a partial model from a complete model, while the latter define points in a partial model that can be modified from the outside. The actual inner structure of the fragment is hidden behind its composition interface: which model elements a certain point maps to and whether it maps to one or more model elements is completely transparent to the user of the fragment. Names of points can be used for reference, e.g., from other patterns.

Fragments are composed by replacing a variation point in one fragment with the contents of a reference point in another fragment (i.e., with a partial model). Reuseware will ensure that such compositions always result in syntactically correct models. Compositions of fragments are expressed in *composition programs.* In addition to composition links, a composition program may also include *settings* through which attributes of elements of a model fragments can be set directly (by providing a primitive value rather than other model fragments). The model transformation generated by our prototype (cf. Fig. 1) produces a composition program for each instance of the abstract DSL metamodel.

Once these preparations have been completed, we can begin using our new abstract DSL. Editors for creating instances of the abstract DSL can, for example, be built using EuGENia [7], which uses an approach similarly based on annotations of the metamodel to generate graphical editors for DSLs. Once an instance of the abstract DSL is created (labelled 'c' in Fig. 1), our prototype transforms it into a Reuseware composition program, which is then executed to produce the corresponding model in the concrete DSL.

4 Conclusions

We have presented a translational approach for defining abstract languages based on more concrete languages. In contrast to an approach where a single monolithic model transformation is constructed from scratch, our approach provides the following benefits:

1. *Simplified construction of abstract languages:* details of the metamodel of the concrete DSML are encapsulated in annotations for the mapping patterns.
2. *Vertical separation of concerns in the model transformation.* The approach separates two concerns in the model transformation: 1) which configurations of concrete-language model elements represent a specific abstract-language model element, and 2) the mapping pattern to use when translating abstract-language model elements into concrete-language model elements.
3. *Use of concrete language tooling for the definition of concrete language configurations.* The approach allows the concrete-language model to be composed from partial template models, each of which can be created and manipulated using standard concrete-language tooling without any need to refer back to the concrete-language metamodel.

References

1. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6) (2000) 26–36
2. Distributed Management Task Force Inc. (DMTF): Common Information Model Standards. http://www.dmtf.org/standards/cim/ (2008) Last visited 28/10/2008.
3. Johannes, J., Zschaler, S., Fernández, M.A., Castillo, A., Kolovos, D.S., Paige, R.F.: Abstracting complex languages through transformation and composition. Technical Report TUD-FI09-08 July 2009, Technische Universität Dresden (2009)
4. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On language-independent model modularisation. Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE (2008) To Appear.
5. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Proc. 1st International Conference on Model Transformation (ICMT). (2008)
6. Louis M. Rose and Richard F. Paige and Dimitrios S. Kolovos and Fiona A.C. Polack: The Epsilon Generation Language (EGL). In: Proc. European Conference in Model Driven Architecture (ECMDA). (2008)
7. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, Fiona A.C. Polack.: Raising the Level of Abstraction in the Development of GMF-based Graphical Model Editors. In: Proc. 3rd MISE Workshop of ICSE. (2009)