Towards Compositional Domain Specific Languages

Andreas Horst, Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

1 Introduction

The deployment of Domain Specific Languages (DSL) and in particular Domain Specific Modeling Languages (DSML) is becoming more and more prominent in various domains. In order to cope with the complexity of the realization of DSLs, common and well-established methods of software engineering such as modularization and reuse need to be adapted and applied for DSLs. This has already been noted in [2] when the emerging DSL era was still closely akin to compiler theory.

As stated in this work, compositionality of DSLs can take place at several dimensions. Various contributions in this field of ongoing research reflect this and only a brief overview is given below. One form of DSL composition is the syntactic embedding such as embedding DSLs in GPLs as described in [3, 4]. In [5] a family of DSMLs are used for the generation of web information systems. There the composition is carried out via the joint usage of several languages each with their own artifacts and hence no syntactic embedding. Other contributions in the area consider the composition of the models expressed in DSLs as a constructive model transformation [1] and examine the effects of DSL composition at the infrastructure level [9] (e.g., syntax aware editors etc.). The DSL framework and workbench MontiCore [7, 8, 10, 11] was designed and realized particularly with respect to compositionality at various dimensions [6, 12].

Compositionality is of special interest if models of different modeling paradigms - and hence expressed in different languages - need to be combined while at the same time retaining their specific semantics. Whenever the different modeling paradigms are integrated, it can be observed that each paradigm is equipped with its own modeling language and that therefore such a paradigm integration is always also a model language composition. This holds for the composition of structural and behavioral languages as well as for the composition of languages with synchronous or real-time communication and event triggered asynchronous models, etc. In the following the dimensions of such compositions are discussed in more detail.

2 Compositional Language Definition

The major rationale of a DSL is its specificity. One could argue that therefore each DSL has to be defined for the specific use case, i.e., the target domain. However, there usually exist common parts being used in various DSLs. This also holds for DSLs serving different paradigms as usually names, types, variables and often signatures are shared.

Thus the DSL development process benefits from a library based approach. Common language fragments can be provided as a library. The definition of concrete DSL then imports, inherits or embeds the required common language definition components (e.g., in form of grammar nonterminals).

Furthermore, features such as checking of context conditions and especially type correctness (i.e. semantic analysis) and other language infrastructure components (e.g., parser, abstract syntax tree (AST)) of a DSL need to be reusable in a reasonable manner. This requires a thoughtful design of the Application Programming Interface (API) the DSL infrastructure is based upon particularly with respect to compositionality.

3 Compositional Modeling

Often it is necessary or at least helpful to decompose a larger description into several artifacts. This capability is the foundation of modularity and reusability and requires the DSL infrastructure to feature processing of models distributed over individual artifacts just as most GPL compilers can process source files in a rather independent and incremental manner. DSL infrastructures hence have to support model artifact dependencies and thus some sort of model path. This mechanism should also allow to incorporate libraries.

For a simple DSL, this compositional modeling can basically be achieved by splitting models and distributing the resulting fragments over several artifacts. Typically the resulting artifacts each encapsulate a specific part of the overall model and respectively exhibit an explicit interface other artifacts can depend on. Therefore DSLs supporting compositional models necessarily have to provide encapsulation, interfaces and imports. This of course greatly impacts the design of the DSLs.

Apart from this rather straightforward case, the composition encompassing models expressed in various DSLs - potentially even with differing modeling paradigms - yields more complex requirements. For this to work, the aforementioned infrastructure (i.e., context conditions, AST, editors) needs to be capable of being glued together to perform all desired and required tasks. The particular challenges of this complex scenario of compositional modeling across language - and potentially even modeling paradigm - boundaries is based on the following dimensions of composition:

- Syntactic: The syntactic dimension describes how the composition of models - in particular expressed in different DSLs - looks like (e.g., textually embedding, split among artifacts, graphical vs. textual etc.).
- Context Conditions: particularly complicated is the dimension of context conditions that spread across the various languages being deployed together; where e.g. types are shared.
- Semantic: The semantic dimension is about the meaning of the individual model fragments and the meaning of their composition. As an example consider the composition of behavioral models (e.g., Statecharts) with structural models (e.g. object diagrams); what does such a composed model express?
- Technical: The technical dimension deals with the tooling infrastructure of the composition. This for instance determines whether the different models can be processed incrementally and/or individually.

– Methodical: Compositionality provides the ability to decompose a problem and to solve it in parts. A good method can and must take decomposition into consideration.
– Organizational: The decomposition of the problem also yields the possibility to have developers solve particular sub-problems in parallel. This allows to organize the team according to the particular composition of the models. Indeed in conventional software engineering - especially for large projects - the organization of the development team is typically based on the problem/product architecture and component structure.

These considerations show that the possibility to decompose a model into several fragments potentially spread across different artifacts and expressed in different languages with clearly defined interfaces greatly influences the development process.

Typically the model composition boils down to the transport of names and related information in the interfaces between the artifacts each encapsulating a part of the composed model. Names are the primary mechanism to refer to when importing some concept from another artifact. Names come with a lot of related information which includes types, method signatures, etc. But most importantly a name needs to be equipped with the kind of model element it represents, e.g., a method, an attribute, a state, an activity; i.e., the respective concept of the DSL. In behavioral languages it is usually necessary to provide some knowledge about behavioral dependencies, such as order dependencies of messages, maximal waiting time before a timeout is executed within the answer awaiting sender, etc. A different example is the composition two models, one being a class diagram and the other an OCL invariant. There the name of a class used in the invariant determines which attributes are valid to be used in the OCL invariant. This name based dependency is independent from the actual form of syntactic composition, i.e., it does not matter whether the two models are expressed in separated artifacts or combined in one artifact using language embedding.

4 Compositional Generators

In practice it is of interest to defer the actual execution of model composition to a later phase of the development or respectively compilation process. This means that while the semantic composition is well known during the creation of the models, the actual composition takes place in a later phase. This deferring of the composition is a major achievement of modern programming languages. Taking the GPL Java as an example, it is well known how classes are combined together, but the actual technical composition - namely the linking - is conducted later on (i.e., there is no source code being copied into a single monolithic source artifact). Instead each source artifact containing a class definition is being compiled independently and only when starting the program these compiled classes are then linked together.

Transferring this idea to the field of compositional modeling and in particular code generators, this means that models are not composed together directly, but the individually generated code will later be linked together. Especially in the case of heterogeneous modeling languages, it is an obvious consequence that a compiler infrastructure is needed which provides a modular compilation unit for each of the individual languages.

The implementation of such compilation units should be independent of other generators, because only then the composition of DSLs and their paradigms can be carried out in a rather flexible way with respect to code generation.

As an example consider a generator for JPA compatible Java implementations of a class diagram. A second generator creates a graphical web information system out of class diagrams which enables users to explore data structures. A third generator adds state to the objects described by Statecharts. All generators should be usable independently but also easily be composable. Now consider that for example the JPA generator creates Java classes with a specific constructor with parameters while hiding the default constructor. If all three generators are to be used together the two other generators have to take the JPA generator's behavior into account in order to produce valid Java code; i.e., they need to use the JPA classes with the correct constructor and in particular cannot assume the availability of the default constructor. Ideally this dependency is handled in a transparent way not obstructing the independence of each generator individually.

Although partially solved for certain instances, a general solution for the problem of a flexible composition of generators is an ongoing research task. It is to be examined in which way generators can be combined using a suitable interface. In the example above, it would be desirable to have the JPA generator provide the information necessary to use the generated domain model classes which in turn can then be correctly used by the Statecharts and the web information system generators (i.e., by the code generated by these generators).

Of course when composing generators, it is not only necessary to have composable interfaces on the generator level, but also to ensure that the generated results are semantically consistent and thus compositional too. Therefore it absolutely makes sense to first solve composition of multi-paradigm models respectively their languages semantically, before this is implemented in generator tools.

References

1. Bezivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M.P., Jouault, F., Kolovos, D.S., Kurtev, I., Paige, R.F.: A Canonical Scheme for Model Composition. In: Verlag, S. (ed.) Proceedings of the Second European Conference on Model-Driven Architecture (EC-MDA) 2006. Bilbao, Spain (July 2006)
2. Bosch, J.: Delegating Compiler Objects: Modularity and Reusability in Language Engineering. Nordic J. of Computing 4, 66–92 (1997)
3. Bravenboer, M., de Groot, R., Visser, E.: MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In: Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05). Braga, Portugal (July 2005), `http://www.cs.uu.nl/~visser/ftp/BGV05.pdf`
4. Bravenboer, M., Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: 4th International Workshop on Software Language Engineering (2007)
5. Dukaczewski, M., Reiss, D., Rumpe, B., Stein, M.: MontiWeb - Modular Development of Web Information Systems. In: Rossi, M., Sprinkle, J., Gray, J., Tolvanen, J.P. (eds.) Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09) (2009)
6. Grönniger, H., Rumpe, B.: Modeling Language Variability. In: Calinescu, R., Jackson, E. (eds.) Foundations of Computer Software. No. 6662 in LNCS, Springer, Redmond, Microsoft Research, Mar. 31- Apr. 2 (2011)

7. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Tech. Rep. Informatik-Bericht 2006-04, Software Systems Engineering Institute, Braunschweig University of Technology (2006)
8. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a Framework for the Development of Textual Domain Specific Languages. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume. pp. 925–926 (2008)
9. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-Specific Languages for Composable Editor Plugins. In: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009) (April 2009)
10. Krahn, H.: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering. Ph.D. thesis, RWTH Aachen University (2010)
11. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer (STTT) 12(5), 353–372 (September 2010)
12. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. Ph.D. thesis, TU Braunschweig (2011)