Towards Model and Language Composition

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

ABSTRACT

Programming languages have one essential advantage over modeling languages: they have a well-defined and understood mechanism for composition that comes along with a good encapsulation of internal details. This is essential when large and complex systems need to be developed. In this extended abstract we discuss the problems of composition techniques for models and compare them to the composition of program components.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: Model Development, Modeling Methodologies, Model Languages

General Terms

Design, Languages

Keywords

Model Composition, Encapsulation, Model Interface

1. CURRENT SITUATION

Composition mechanisms and in particular the possibility to decompose a system into smaller components for independent development is an essential capability that today's programming languages –like Java– provide. Modularity and composition mechanisms are essential for the ability to develop large, complex systems in distributed teams and in parallel [11]. They allow to reuse components from libraries and they are the basis for efficient, incremental compilation, generation and also many forms of analysis.

It is interesting to analyze how programming languages deal with composition. Formally, composition is a mechanism that takes two given artifacts A, B and composes them based on their interfaces, but encapsulates all internals not visible in the interface. This is a core principle of object orientation as well as of function theory. In programming languages we have an important coincidence between the *component* (resp. class or module) that has an explicit interface and an encapsulated implementation and the *artifact* (i.e. file) that contains that component. While we logically compose the components, the tools de facto compose the artifacts. Furthermore, the composition is not executed on the sources, but very late on the compiled binaries. So composition remains a logical form of combining components that consistently maps to the late binding of the binaries. As a consequence compilers can incrementally recompile from changed sources, without touching the whole project. That way compilers are efficient enough to actually allow Agile Methods like XP [2] and Scrum [12].

So what is different in the modeling domain? Indeed we have quite a number of differences:

- We do not deal with a single form of model, instead we have *heterogenous, composed languages*, such as the UML, because modeling uses various viewpoints for specific kinds of information. These viewpoints are not capable of describing all aspects of the system. Heterogeneity is a natural consequence.

- Some of the modeling languages do have a notion of *interface*. For example state machines typically provide input and output as an interface, while states remain locally encapsulated. Other modeling languages, such as class diagrams, do not provide a useful notion of interface for the diagram at all. The idea of interfaces is not very elaborated in the modeling domain. It is also important to distinguish between the interface of a model and the interface of the elements that the model describes. This can be easily mixed up, for example in class diagrams.

- Current modeling tool infrastructures try to load and manage all models in their space at the same time. This *scales up* only to a certain extent and makes modeling tools very slow, such that agile development is not possible anymore. *Incremental development* does not play a major role as late binding of generated code is not applied.

- The classical notion of *artifact* is not very present in the modeling domain, as many tools only deal with one big model that they store in the database. As model elements are connected very much, it is impossible to develop reusable, independent modeling artifacts and

[Rum13] B. Rumpe
Towards Model and Language Composition.
In: GlobalDSL '13 Proceedings of the First Workshop on the Globalization of Domain Specific Languages, ACM New York, NY, USA ©2013, pages 4-7, ISBN: 978-1-4503-2043-6.
www.se-rwth.de/publications

libraries. Model libraries with reusable assets only exist, if interfaces of models and references to these interfaces become part of the modeling language.

- *Reuse of models* is only in its infancy. First approaches to collect models like ReMoDD [1] experience problems of reuse because of missing techniques to incorporate foreign models, but also due to a lack of a standard for model interchange between tools.

These are major inhibitors that have not allowed us to come up with solid and generally accepted notion of composition for models and modeling languages so far. Instead a number of approaches have suggested to use techniques like model weaving and merging [4, 7], which are well known from aspect oriented programming, but disregard the idea of encapsulation.

However, as we see more and more projects using modeling technologies, it is time to actually refine and enhance composition concepts on the modeling languages that we will use in the future. In the remainder of this paper, we therefore classify a number of questions needed to be handled to achieve model composition and take a look at the notion of interface.

2. FORMS OF MODEL COMPOSITION

Composition is an often used term in Computer Science. It dates back to mathematical function composition and was reinterpreted by object composition that basically deals with data structures and also data type composition as a combination of both in the abstract data type domain.

we define *model composition* as the derivation of a model $C = A \otimes B$ from two base models A and B using an appropriate *composition operator* $\otimes$. A core element of composition is the existence of two artifacts A, B with a sound meaning, precisely defined interfaces and internals that are not to be exhibited to the outside. The composition $C = A \otimes B$ glues both artifacts together on their common interfaces, such that they have a combined meaning.

We can distinguish between the following forms of composition and the questions they try to tackle:

Syntactic composition:
How does $A \otimes B$ actually look like?

Semantic composition:
What does $A \otimes B$ mean?

Methodical composition:
How to develop the models A and B such that the resulting composition conforms to the desired requirements?

Organizational composition:
How to decompose the development tasks into parallel jobs for A and B when in a large team?.

Technical composition:
Is incremental, artifact-individual compilation and late binding of the code from A and B possible?

As an aside it is necessary to clarify whether composition is commutative ($A \otimes B = B \otimes A$) and associative (order is irrelevant in $A \otimes B \otimes C$). That means we do not have to explicitly determine the order of composition, but can leave this open to generators and compilers.

2.1 Syntactic Composition

Syntactic composition is usually applied, when A and B are used as views and are syntactically integrated and woven together to come up with a complete model C. As a consequence a tooling infrastructure even the smallest change of a single model element in one of the views leads to a complete re-generation and thus takes an unnecessary long time.

It is then also quite common to allow the user to read and modify the composed model C. This is actually a bad idea, as the original models are then out of sync and can never be adapted and reused again. This is quite similar to allowing programmers to adapt object code.

2.2 Semantic Composition

Semantics as understood in [6] deals with the question, what a composed model means by deriving that from the semantics of the composition operator and the semantics of the individual models. Thus having a semantics for the composition allows us to uniquely understand what the result of a composition is. Different tool vendors can then implement their own, optimized, but semantically equivalent versions of the composition.

Please note that semantic composition does not mean that the composition is implemented exactly that way. Instead it is used to understand, but not having to explicitly execute the composition.

2.3 Methodical Composition

Composition of individual components is actually mainly about being able to *decompose* the overall problem statements early in the development process. It is important to be able to do the decomposition of the requirements during the process in a robust and stable way such that the resulting composition of solutions does what is desired. Among others, coherence and decoupling guidelines play a role.

2.4 Organizational Composition

When it is clear how to decompose a (not yet developed) model, we should then be able to organize our team and project in such a way that the decomposed components can be developed independently, but ideally with a continuous integration process behind. Agile methods show how that works using early testing. Modeling technologies can add early analysis and consistency checks.

2.5 Technical Composition

When we know what composition actually means and when we are able to refine our modeling languages in such a way that interfaces become an intrinsic part of the models, then we should be able to delay the composition of solution models after code respectively tests are generated.

This does not only create a tremendous speed up, as small changes than just lead to small re-generations. It also allows

us to come up with pre-compiled code from model libraries that can be used without the need to access to the source models. This was a major breakthrough for programming languages and is essential for further success of modeling technologies.

One particular problem to be solved is the lack of coincidence between the artifacts' name and place in the directory versus the name of the components described in the artifacts.

3. LANGUAGE COMPOSITION VERSUS MODEL COMPOSITION

Above we strongly argue that model composition does not mean to execute the composition on an syntactic level.

This is different, when we take heterogeneous languages into consideration. In modeling it is often the case that even if an artifact belongs to one modeling language, such as a class diagram or a statechart, we find embedded elements from other modeling languages, such as OCL expressions, code statements and others.

We do not regard this to be model composition in a strict form, as there are neither a composition operator nor have the sub-artifacts ever been developed independently. Instead we see a single model artifact belongs to a composed language.

A *modeling language composition* is a combination of sub-languages into one complete language, where the individual sub-artefacts adhere to their sub-languages and the complete artefact derives its syntax and semantics from the composed sub-languages [13].

Interestingly language composition shares many of its problems with model composition such that the above questions (with the exception of syntactic composition) equally apply.

4. INTERFACES OF MODELS

Within our ongoing work, we have collected experiences from more than 30 domain specific languages, a larger subset of the UML that contains sequence diagrams, statecharts, class diagrams and object diagrams and the OCL, as well as the programming language Java. We developed the hypothesis that interfaces between modeling artifacts always rely on names (or their anonymous colleagues, the identifiers). We are still searching, but are rather sure that this will at least be valid in a majority of cases.

Interfaces are imported, exported and partially also their elements passed through (imported and re-exported). An artifact may export several interfaces. We know e.g. from programming languages that an interface can be for sub-classes, public or local to the package.

An interface contains names and these names consist of different *kinds of elements*. In programming languages we do have the kinds: class, type, attribute, method, constant. The UML e.g. additionally provides many more kinds like state, message or activity. Domain specific languages may have domain specific kinds of names. It would be wrong to describe interfaces only on the level of a target programming language. A more general mechanism of interface with different kinds of names is necessary and realized in MontiCore [10, 9, 8].

For *heterogeneous languages* we need *mappings between the interfaces*, as it may be that special kinds of elements are exported by one language, but unknown by another. A good example for this is the kind "*state*" in state machines that does not exist in Java. But we also need to be aware that this form of composition using mappings between the interfaces of heterogeneous languages is not necessarily unique. For example language variability [5, 3] allows us to map state T to programming constructs such as a constant T of an enumeration, or into a class with the same name as indicated by the state pattern or a method *isinT()* to check the status. In OCL state T needs no mapping as OCL is state-aware though built-in function *oclIsInState()*.

5. SUMMARY

Composition of models in heterogeneous modeling environments is important for an even more successful use of modeling technology in software development projects. Composition of models is, however, currently not very well elaborated, partly because lots of good ideas from the programming domain have not been carried over to the modeling domains yet.

Composition influences how languages will look like. This might lead to an enhanced version of the UML as well as improved tooling infrastructures for the UML and domain specific languages.

6. REFERENCES

[1] ReMoDD The Repository for Model-Driven Development, 2013. http://www.cs.colostate.edu/remodd/, [Online; accessed 10-June-2013].

[2] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison-Wesley, 1999.

[3] M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within modeling language definitions. In *Model Driven Engineering Languages and Systems (MODELS) 2009*, number 5795 in LNCS, Denver, Colorado, USA, 2009. Springer.

[4] M. D. Fabro and P. Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *The 22th Annual ACM SAC, MT 2007*, 2007.

[5] H. Grönniger and B. Rumpe. Modeling Language Variability. In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software*, number 6662 in LNCS, Redmond, Microsoft Research, Mar. 31- Apr. 2, 2011. Springer.

[6] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[7] F. Heidenreich and H. Lochmann. Using Graph-Rewriting for Model Weaving in the context of Aspect-Oriented Product Line Engineering. In *First Workshop on Aspect-Oriented Product Line Engineering (AOPLE 06)*, Portland, Oregon, 2006.

[8] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and

S. Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, number 4530 in LNCS, pages 99–113, Haifa, Israel, June 2007. Springer.

[9] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[10] H. Krahn, B. Rumpe, and S. Völkel. 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.

[11] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[12] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[13] S. Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. PhD thesis, TU Braunschweig, 2011.