Configurable Formal Methods for Extreme Modeling
Position Paper

Uli Fahrenberg and Axel Legay
IRISA / Inria Rennes

Abstract. Reliable model transformations are essential for agile modeling. We propose to employ a configurable-semantics approach to develop automatic model transformations which are correct by design and can be integrated smoothly into existing tools and work flows.

Model management is an essential activity in a model-driven development process. This is especially true in an agile context where models are continuously refined, iterated and combined.

It is a common misunderstanding that development by stepwise refinement, or use of component algebras, requires using a highly planned and waterfall-like development process. See for example the following quote:

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification. [30, p.32]

We will argue below that methods from model management and formal system development, when combined with and inspired by approaches originating in the area of formal interface and specification theories, can play an important role also in an agile context.

Within the context of model-driven engineering, automated model transformations such as merging, differencing and slicing are of great importance. This is especially true in the industrial context where models can easily get so large that the engineers only see them through viewpoints, or slices; indeed, an explicit system model may not even exist, so that the general model is only implicitly given as a collection of viewpoints.

When the system model is so complex that no single engineer has a comprehensive view, it can be very challenging to ensure correctness of an applied model transformation by inspection. (Even when the system model is less complex, ensuring correctness by inspection may be a difficult and error-prone process and
require advanced tooling.) The use of model transformations which are correct-by-design, or at least checkable-by-design, hence becomes increasingly important. This point of view has also been argued in [1, 6, 24, 27].

A good example is given by one of the case studies in the MERgE ITEA2 project [28]. This consists of a large system model with multiple (more than twenty) viewpoints, each detailing a different aspect of the model. The involved engineers are only working with the model through these viewpoints, as the whole model is too complex to be worked with directly. Now when a model transformation is applied to the model (e.g. a new subsystem is added through model merging), the engineers can inspect at their respective viewpoints whether, locally, the model transformation has been applied correctly. But can we be sure that this implies that the transformation is also globally correct? Can we design a procedure which allows such kind of local-to-global reasoning?

To put it succinctly, it is an important problem in model-driven engineering to ensure that model transformations are semantically correct or that, at least, their semantic correctness can be inferred by a combination of slicing and local inspection.

One basic model transformation is the one of differencing, i.e. assessing differences between models. This is an important ingredient in version control and essential in three-way merging, but can also be applied, more elementarily, to inspect related models for their common points and differences. Semantically correct differencing procedures, for class diagrams and activity diagrams respectively, have been proposed in [24–26]. However, these procedures rely on a complete, formal semantics of the modeling formalism in question, which for most cases is unrealistic: in practice, engineers use modeling tools which do not have a formal semantics, or where an existing formal semantics is too complex to be practically useful.

It is our point of view that any “bottom-up” approach to correct model transformations which uses a complete formal semantics as a starting point, such as the above-cited, is of doubtful use in practice. In practice, engineers develop models according to an intuitive understanding of how things work, and not according to a complete formal semantics (if indeed it exists at all).

Apart from being correct-by-design or checkable-by-design, we have also argued in [18] that it is important that model transformations return an object of the same type as the inputs. Hence, the merge of two class diagrams, for instance, should again be a class diagram, the difference of two feature diagrams should again be a feature diagram, etc. This allows developers to visualize the transformation and to manipulate it using the usual tools for working with models: the transformation integrates smoothly into existing tools and work flows. Figure 1 shows an example of such a work flow which necessitates that the difference between two models is again a model of the same kind.

[24–26] propose semantic difference operators for class diagrams and activity diagrams, respectively, however, their approaches are enumerative in nature: their output is a (potentially infinite) list of object models which witness the difference between the input diagrams. The output language is thus different
**Fig. 1.** Example of a work flow where the changes (“diff”) between models $M_1$ and $M_2$ are automatically applied (“merge”) to $M_3$. Note that a simplistic “diff” is not always the correct approach; it can be important to e.g., tell the difference between renaming an entity and a deletion followed by an addition.

from the input language, which makes it impossible to integrate their tool into a standard work flow without additional processing steps.

Within the subject of interface and specification theories in formal methods, semantic model transformations exist for many types of low-level behavioral models [3–5, 7–9, 11–14, 20, 22, 23]. In recent work, we have developed a generalization of these approaches in which the semantics of models is configurable [2, 16, 17]: When models and specifications contain quantities such as timing information or resource use, the precise behavior of model transformations depends on the type of quantities and on the application. Hence, the precise definitions and properties of model transformations depend on the quantitative semantics, and our generalization offers a generic way of configuring the transformations and properties according to the semantics.

Similarly in spirit, Maoz et.al. have in [10, 27] introduced techniques for semantically configurable analysis of some high-level models, viz. class diagrams and scenario specifications. They use feature diagrams [21, 29] for configurability, so that the analysis depends on the selection of features.

We have in [15, 18] introduced semantically correct model transformations for feature diagrams and class diagrams, see Figures 2 and 3 for some examples. These operators’ return types are the same as their input types, so that they can be integrated smoothly into existing tools and work flows, but they rely on a complete formal semantics. We believe that these approaches can be combined with the configurability of [10, 27] to yield model transformations which are automatic and correct-by-design, yet flexible enough to be practically useful.

**Conclusion**

We propose to employ a configurable-semantics approach, using feature diagrams, to develop automatic model transformations which are correct by design and can be integrated smoothly into existing tools and work flows. Such model transformations are important for agile modeling methods.
As an example, consider the following scenario: An engineer wants to perform a three-way merge of two models which have evolved from a common ancestor. She will first use her development tools to attempt an entirely syntactic merge, which will detect some conflicts which are inessential because they only amount to syntactic differences while being semantically equivalent (given the developers’ intuitive understanding of the semantics). These she can easily detect, and she can adjust her merge tool to take them into account (hence applying a partial semantics). However, due to semantic effects which propagate through the models, also the reverse may happen: there may be semantic conflicts which go undetected by the syntactic approach. These are more difficult to detect and require that even a merge which is syntactically correct be carefully checked by applying possibly several different semantics.

References

Fig. 3. Two feature diagrams and an over-approximation of their difference, cf. [18].
18. Uli Fahrenberg, Axel Legay, and Andrzej Wąsowski. Make a difference! (Semantically). In Whittle et al. [31], pages 490–500.
27. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically configurable consistency analysis for class and object diagrams. In Whittle et al. [31], pages 153–167.