Refactoring of Automotive Models to Handle the Variant Problem

Cem Mengi and Manfred Nagl

Software Engineering, RWTH Aachen University, Germany

1 Introduction and Motivation

Models and model-based languages are more and more used in the *automotive* domain [1] to define artifacts for software (requirements, software and hardware architectures, code structure and behavior) to run in a car. Models and code reflect a remarkable complexity, due to safety critical properties and the huge amount of possible variants [2].

The big number of *variants* comes from simple or elaborated functionality, different realizations of functionality, combination of functionality, integration of proprietary solutions, country-specific legality constraints, and alike. This presentation concentrates on *Simulink* models [3], which are an essential part of the model structure describing automotive software.

Especially, we show how to use *refactoring* [4] in order to express commonalities and differences of models and, therefore, facilitate reuse and extensibility as essential quality measures. Even more, refactoring is also the means to handle the variant problem. The presentation is based on a chapter of [5]. It remains on an informal example by example basis.

2 The Approach

Software variants in the automotive domain are usually modeled incrementally and evolutionary by *copy-paste* methods. These methods are *unsystematic* and result in *unclear* designs, where abstraction is missing. Especially, due to the increasing complexity, common parts of the models are very hard to be identified.

The consequence is a decreasing quality according to measures like reusability, extensibility, modifyability etc. *Refactoring* is appropriate to overcome the drawbacks. It consists of two main *activities*:

1. Differencing Simulink models to identify common and variable parts.
2. Restructuring the models with variability mechanisms to improve quality, as they now express commonalities and differences.

While tool-support for differencing is either *automatic* or *interactive*, restructuring is *rule-based* and could be supported by tools. In the following Section 3, we describe the differencing of Simulink models. After that, Section 4 deals with restructuring.

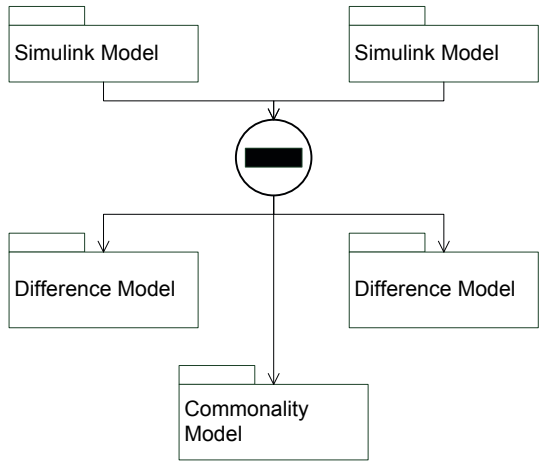

Figure 1: Overview of the differencing activity

3 Differencing Simulink Models

Figure 1 illustrates the differencing approach. The differenciator function gets two Simulink models as input and puts out three models, one *commonality model* and two *difference models*. The commonality model consists of Simulink blocks and connections, which are *common* or *similar* in both input models. Furthermore, it also comprises the *points of variation*. These variation points are defined by the difference models. Therfore, they include solely *variant-specific* details.

In simple situations, differencing is done automatically. More complicated situations are handled interactively.

Having this information, the original Simulink models can now be restructured in a way, that reusability and exensibility can be significantly enhanced.

4 Restructuring Simulink Models

Our *variability mechanisms* are means to implement variation points in Simulink. Mechanisms are `If Action Subsystems`, `Switch`, `Model Variants` and `Variant Subsystems`. It is important to know about the properties of these mechanisms before using them for restructuring models, as wrong decisions can lead to further expensive restructuring.

We have analyzed and evaluated the above vari-

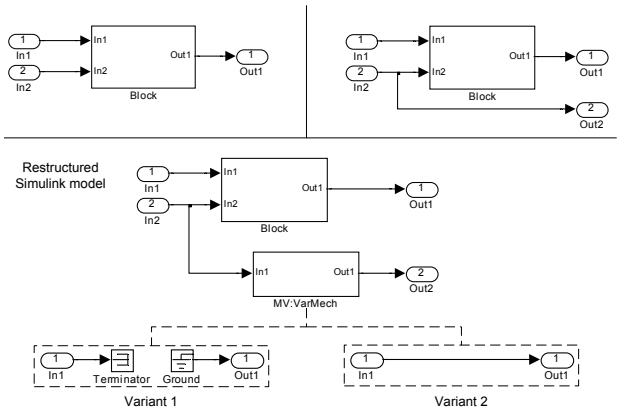

Figure 2: Example of a restructuring rule

ability mechanisms by considering suitable underlying control constructs, the operating mode, and their advantages as well disadvantages on the basis of various criteria [5]. The outcome of this has shown that `Model Variants` and `Variant Subsystems` can be favored. Therefore, we recommend to restructure Simulink models by mainly using one of these two variability mechanisms.

To support the restructuring activity, we adopt a rule-based approach, where each rule defines a *recommended action* for a detected variation point. The rules cover the *interfaces* of (root/inner) models and blocks, *block types* and *connections* between blocks. They can be applied for *simple* as well as *complex situations*.

Figure 2 shows a simple (and non realistic) example of a situation to be restructured. Both upper left and right Simulink models share a lot of commonalities. The only obvious variation point, which would also be detected by the differentiator function, is the out-interface.

An *appropriate rule* for this variation point can be formulated as follows:

If there is a variation point at the out-interface, then do the following in the restructured model:
1. Model the *maximum out-interface*.
2. Apply `Model Variants/Variant Subsystems` to all variable ports in order to *control the data flow* for each variant:
   - 2.1 The data flow of a port, not existent in a variant, is terminated with a `Terminator`-block. To achieve adaptability, a dummy signal is generated with a `Ground`-block and forwarded to the variable port. (Both simulate an empty connection.)
   - 2.2 The data flow of a port, that exists in a variant, is just forwarded.

In this way, we have formulated nine *basic rules*, which can be combined to more *complex rules* in order to handle complex variation points. Doing so, the restructured models achieve a quality level, where *reusability is improved* (commonalities are captured and modeled once) and *extensibility is enhanced* (variability mechanisms).

5 Extensions

The example above was one model with two different occurrences of variants. Of course, also *multiple variants* of that situation work in the same way. Even more, the situation can be generalized such that two or more *dependent* situations are handled: As example we sketch in our presentation the car access system (with a simple or an elaborated variant) and its relation to the comfort system (adjusting the seat, the mirrors, the heating/ air conditioning), again in a simple and an elaborated form.

The approach can also be used for *other models* as function networks, software architectures, code fragments etc. On any of these models also dependency situations can be handled. In [5] the variant problem is studied on function network, Simulink, and code level.

6 Summary

We showed that one key to the *variant problem* is to be able to *model commonalities* and *differences* of models, dependent models, and their combinations. The approach can be used on different levels (function networks, Simulink, software architectures, code), so for all relevant modeling artifacts used in automotive software.

This variant approach can be applied in a *refactoring* mode and thereby finding commonalities and differences of given models, in a *top-down* approach where we start with commonality/ difference modeling which is furthermore refined, or in a combination of both. Therefore, also round-trip engineering is covered.

References

[1] Schäuffele, J. and Zurawka, T.: Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen. *Vieweg*, 2006

[2] Clements, P. and Northrop, L. M.: Software Product Lines: Practices and Patterns. *Addison-Wesley*, 2007

[3] `www.mathworks.de/products/simulink/`

[4] Rech, J. and Bunse, C.: Model-Driven Software Development: Integrating Quality Assurance. *Idea Group Reference*, 2008

[5] Mengi, C.: Automotive Software - Prozesse und Variabilität. Ph.D. Thesis, to appear