Automatic Evaluation of Modelling Rules and Design Guidelines

Tibor Farkas, Christian Hein, Tom Ritter

Fraunhofer Fokus, Kaiserin-Augusta-Allee 31,
10589 Berlin, Germany
{tibor.farkas, christian.hein, tom.ritter}@fokus.fraunhofer.de

Abstract. In order to get high-quality software systems high-quality models are needed. Modelling rules and design guidelines can help to improve the quality of a model. This paper presents a flexible solution for phrasing and evaluating modelling rules and design guidelines for models by using the Object Constraint Language as the target environment to achieve a higher quality of models. The presented solution is based on Open Source tools like OSLO and describes how it is applied in an automotive domain.

Keywords: design guidelines, modelling rules, guideline packages, Object Constraint Language

1 Introduction

In a model-driven development environment the quality of models is crucial for the quality of the resulting software. Techniques like model transformation and code generation are used to produce automatically or semi-automatically software systems from design models, which are possibly platform independent. Since models are of great importance every possible effort should be made to ensure the best possible quality of the model. Therefore the modeller, who designs a system, has a difficult task and takes high responsibility.

In traditional development processes program code like C++ has to follow specific coding rules and style guidelines. Organisations like NASA have their own individual set of style guidelines [13]. In model driven development processes, modelling rules are part of the metamodel and define the correctness of models. Modelling rules are expressed in either natural language or formal language. Design guidelines are similar to coding style guidelines since they do not really constrain the correctness of a model but they help to improve the quality of the system in a more indirect way. To follow a set of design guidelines usually helps to improve the clearness and the readability of a code fragment or a model. It helps to avoid misinterpretations and makes it easier maintaining such artefacts.

In the past, many approaches dealt with the automatic checking of coding rules. Such checks usually worked on the source files (i.e. C++ programs) [14]. Today, some approaches already exists which try to automate the checking of design guidelines in model driven development process. This paper presents experiences

made with the application of checking modelling rule and design guideline in the automotive domain.

2 How to Check Modelling Rules and Design Guidelines in an Automatic Way

Design Guidelines could be used to achieve a higher quality of models. We define a quality of a model as follows: The set of properties of a model which are of interest in a specific situation or scenario. Each of the properties has a value space (dimension) in which an ordering relationship is defined. For example naming conventions for attributes or methods could contribute to the clearness of models. We could define the order relation in a way that a model which adhere such conventions is in a higher order than a model which fails these conventions. Furthermore, a model should contain only elements which are needed and which reflects element of the system that should be built, not more but also not less. Models which adheres naming conventions and also only contains needed elements are again in a higher order than models which obeyed only naming conventions.

It is difficult to verify the quality that a model should include only needed elements. For instance a UML2 [1] model could contain an actor which is not in a relation to a use case. The actor as a model element is needless and can lead to confusion if the model is read by other developers. To avoid such situation it is useful to check models with respect to such design guidelines at certain points in time, e.g. whenever a model gets a development tag.

In a standard modelling process the designer of a model has to consider implicit and explicit design guidelines. Such an explicit guideline could indicate naming conventions like all class names have to start with a capital letter. An implicit rule could be that the designer designates names meaningfully. With a set of such design guidelines it is possible to make the model more readable and plain.

Design guidelines influence the system in an indirect way. The developed system has no preference whether a class name starts with a capital letter or not. Therefore design guidelines must not be strict. Hence if a model breaks a guideline this must not result in an error, in fact it could result in a kind of a warning.

Modelling rules are strict. They do constrain the concept space that is defined by a metamodel, which is typically described by a using metamodel language like MOF [11]. Models that do not conform to modelling rules can be considered as correct instances of the metamodel. For example a UML2 model is incorrect when it contains an actor which has no name.

Modelling rules and design guidelines are formulated at metamodel level. Without an automatic verification the adherence to design guidelines and modelling rules isn't easy to check for big models. A method to achieve these goals is the specification of rules with a formal language, because one of the advantages of a formal language is that rules could be check by a corresponding tool automatically. There are a lot of formal languages which are useful for this purpose. In our approach we take the Object Constraint Language (OCL) [2] [4] to specify design guidelines and modelling rules for models. In difference to traditional formal languages which are mostly

applied in academic world OCL was developed to be applicable for a large number of users. OCL is a formal specification language without side effects, program logic or flow control.

Modelling rules and design guidelines as OCL constraints could be evaluated with a corresponding OCL tool automatically. For our case study which is described in the next section we have taken OSLO [3]. OSLO is an acronym and stands for Open Source Library for OCL. It is based on the Kent OCL library. Likewise the Kent OCL library, OSLO supports the evaluation of OCL constraints against metamodels. This feature is essential for verification of modelling rules and design guidelines.

3 Experiences in automotive domain

In the automotive industry the development of embedded system software needs modelling, simulation and analysis of dynamic system behaviour in the early design phases by the OEM (Original Equipment Manufacturer). Rapid prototyping technologies are used to develop and optimize new control concepts. Design tools enabling these capabilities such as MATLAB/Simulink [6] help developers to virtually design embedded controllers directly in various block diagrams. Simulink as an Add-on product to MATLAB is a platform for multidomain simulation and model-based design of dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that let developers accurately design, simulate, implement, and test control, signal processing, communications, and other time-varying systems. It also supports linear and nonlinear systems, modelled in continuous time, sampled time or a hybrid of the two. The following figure 1 presents a simple engine timing model created in Simulink. The model describes the simulation of a four cylinder spark ignition internal combustion engine.

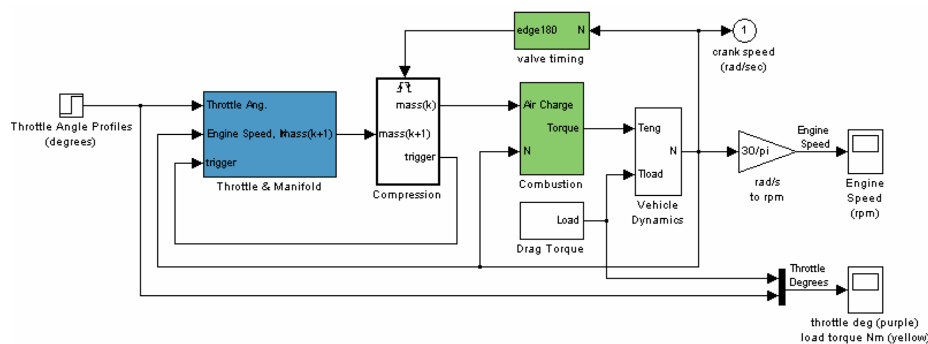

Figure 1 Engine timing model created in Simulink, [7]

Real-time code is generated from block diagrams and can be automatically implemented on flexible prototype hardware. In a closed loop, developers can record time histories in real time, change parameters on-line, and run various scripts. Perform these tasks during early development phases of the V-Modell [8] could lower overall development costs.

However, due to the possibility of a huge range of modelling capabilities provided in Simulink, it makes it hard for developers to stick to a standard notation and corporation-wide policies of models. As a result many different designed models exists which contingently does not correspond to each other. On that reason the MathWorks Automotive Advisory Board (MAAB) [9] was established to coordinate feature requests from several developers in the automotive industry. The initial advisory board involved Ford, DaimlerChrysler, Toyota and many of the major automotive OEMs and suppliers. To give developers of automotive control systems a standardized notation and to be able to assist with achieving system integration without problems the MathWorks Automotive Advisory Board developed the so called: *Controller Style Guidelines For Production Intent Using MATLAB, Simulink and Stateflow* [10]. The guideline document contains up to one hundred guidelines that are related, similar or even (seemingly) contradicting each other. A sample of such a guideline is shown in figure 2:

4.1.2. jm_0001: Prohibited Simulink standard blocks inside controllers

ID: Title	**jm_0001: Prohibited Simulink standard blocks inside controllers**
Priority	Mandatory
Scope	MAAB
Automation	Possible
Prerequisites	
Description	**Controller models must only be designed from discrete blocks.** **Sources are not allowed:** Signal Generator Step Ramp Sine Wave Repeating Sequence Discrete Pulse Generator Pulse Generator Chirp Signal Clock Digital Clock From File From Workspace Random Number Uniform Random Number Band-Limited White Noise **Continuous blocks are not allowed:** Integrator Derivative Memory Transport Delay Variable Transport Delay State-Space Transfer Fcn Zero-Pole **Other blocks. which are not allowed:**

Figure 2 Guideline to avoid the usage of specific blocks inside controllers

Other guidelines for instance describing defaults in detail for adjustment in font-name, font-size, font-weight, fore- and background colour, settings of blocks or window appearance (e.g. All Simulink windows should have a white screen colour.).

This section presents an applicable approach on how to automate the evaluation of complex design guidelines such as the MAAB for the development of automotive control systems with MATLAB/Simulink. From metamodels that are defined using the Meta Object Facility (MOF) [12] it is possible to automatically derive model repositories where models conforming to that metamodel can be stored and accessed through standardized interfaces. To exactly evaluate structural consistence and constraints the Object Constraint Language (OCL) can be used on instances of the

metamodel. The Model Driven Architecture (MDA) [11] propagated by the Object Management Group (OMG) prescribes a set of model artefacts to be used along system development, how those models may be prepared and their relationships. It is an approach to system development that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. The Meta Object Facility (MOF) together with Model Driven Architecture (MDA) has shown that metamodel based language definitions impose new ways of defining language semantics as a separation of syntax and semantic concept space, of integrating languages via a common metamodel base and of generating and deriving models from other models via model transformers. In that case it is necessary to have a Simulink meta model developed and to automatically convert via model transformers a Simulink model as an instance of the source metamodel and to check that constraints are fulfilled. Figure 3 shows a simple subset of the Simulink meta model:

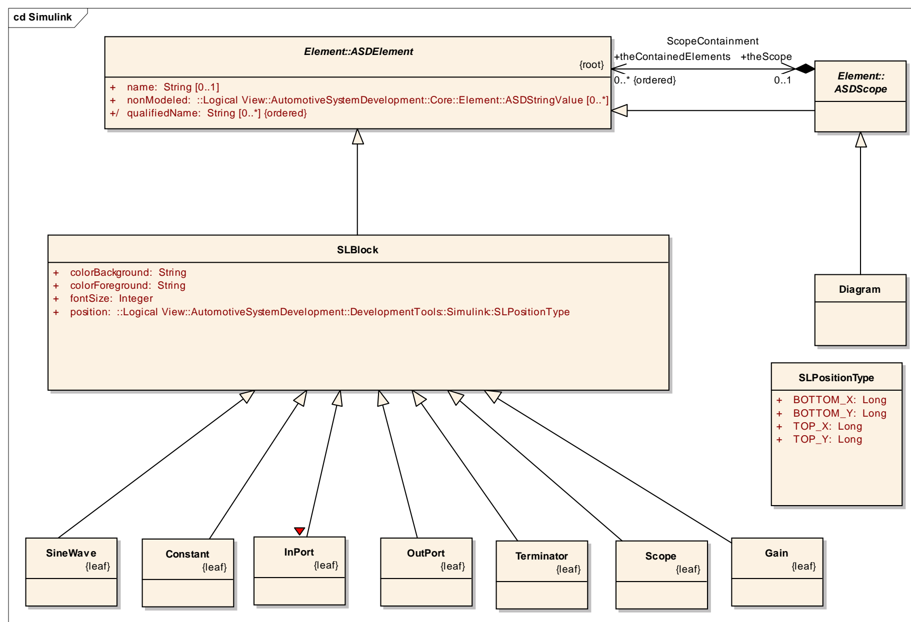

Figure 3 Subset of the Simulink Metamodell depicting design aspects

A Simulink instance model (the engine timing model) that does not correspond to some of the MAAB defined guidelines is presented in the following figure 4:

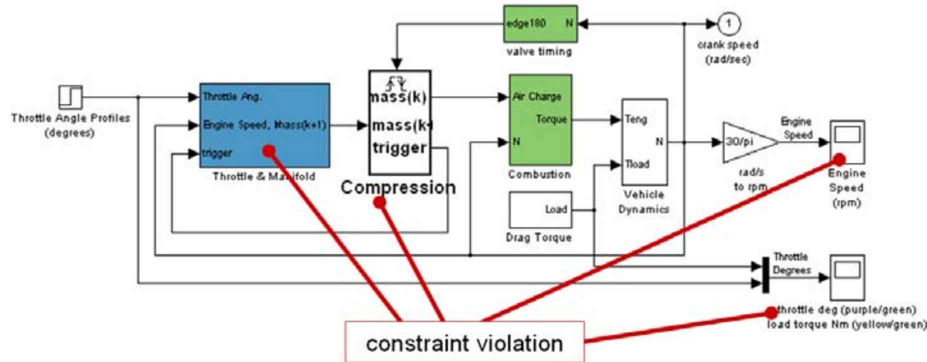

Figure 4 Engine timing model with constraint violation.

The engine timing model shows exemplary four violations against the corporation-wide policies taken from the *Controller Style Guidelines For Production Intent Using MATLAB, Simulink and Stateflow*:

Violation 1:
The *Throttle and Mainfold* (type of Subsystem) violating the default background color setting: white.

Violation 2:
The *Compression Block* (type of Subsystem) violating the specified font-size: 10.

Violation 3:
The usage of the block *Engine Speed* (type of Scope) is prohibited.

Violation 4:
The designation of *Throttle Deg* (type of Scope) exceeds the allowed length.

The given model as an instance of our meta model (figure 3) lets us define corresponding OCL formed constraints to the given policies in plaintext.

OCL evaluation on violation 1:
All elements within a model must have the same common background colour: *white*.

```
context Diagram inv:
theContainedElements->select(b   |   b.oclIsKindOf(SLBlock)
and b.oclAsType(SLBlock).colorBackground<>'white')
```

OCL evaluation on violation 2:
All elements (blocks, signals, labels...) except text annotations within a model must have the same common font style and font size. The specified font-size is *10*.

```
context Diagram inv:
theContainedElements->select(b  |  b.oclIsKindOf(SLBlock)
and b.oclAsType(SLBlock).fontSize<>10)
```

OCL evaluation on violation 3:
The usage of the block *Scope* is prohibited.

```
context Diagram inv:
theContainedElements->select(b | b.oclIsTypeOf(Scope))
```

OCL evaluation on violation 4:
The name of all elements should not exceed the maximum length of 31 characters.

```
context Diagram inv:
ASDElement.allInstances()->forAll(i | if i.name. oclIs
Undefined() then true else i.name.size() <= 31 endif )
```

The aim of this approach is to demonstrate that it is applicable to define constraints through the usage of the Object Constraint Language for an automated evaluation of automotive specific development models. For this purpose MATLAB/Simulink metamodels are introduced and presented in brief. Using OCL Syntax, Semantics and Tools like OSLO with the usage of Simulink metamodels enables the addition of semantic and consistency information in an exact and unambiguous manner.

4 Summary

Experiences we made in our automotive case study show that it is possible to verify modelling rules in an automatic way. But one of the difficulties we had was the specification of the OCL constraints. It was easy to express the guidelines and modelling rules in a natural language but using OCL and working on the specific MATLAB/Simulink metamodel was sometimes hard. Another problem was that some guidelines are ambiguous or not precise. For instance: “All model components should have a meaningful identifier given”. It is difficult to transform such a guideline from a natural language notation into a formal language notation like OCL.

In our experience it is a fact that OCL is hard to learn by contrast to the formulated goal of OCL. Therefore the acceptance in industry world is comparable with the acceptance of traditional formal languages like Z. But nevertheless it is difficult to realise an automatic evaluation of modelling rules without a formal specification of such rules. This means that OCL experts and domain experts have to work together to formulate OCL expression which corresponds to the defined design guidelines. The result of such cooperation should be canned into OCL packages which can than be

used by non-OCL experts to accomplish the actual test. It is conceivable that one package may deal with name conventions and another package may contain modelling rules which are responsible for semantic guidelines. The user would only get information about the packages and its purpose but nothing or few about the concrete formal descriptions with OCL.

The definition of packages raises a new question, which we currently investigate and due to space limitation is not described in detail here: How could such packages reused in the same or in other domains? In particular naming conventions may be the same or they may be similar in various different domains. For that purpose a package should have parameters for tailoring the package for a specific purpose.

Another topic which is currently under investigation and which is closely related to the work presented in this paper is the computation of model metrics. Model metrics are indicators which allows for example to quantify the quality of a model. For instance the average number of model elements contained in a package maybe part of a model metric. OCL as a formal language can be used to formulate expressions that help to compute more complex model metrics than guidelines, because guideline do only have the Boolean dimension (i.e. true or false). In contrast to design guidelines model metrics have more complex dimensions such as real numbers. This enables a more granulated ascertainment of model qualities.

References

1. UML – Unified Modelling Language (UML) Specification: Infrastructure. http://www.omg.org/docs/ptc/04-10-14.pdf
2. OCL – Object Constraint Language 2.0 Specification. http://www.omg.org/docs/ptc/05-06-06.pdf
3. OSLO – Open Source Library for OCL. http://oslo-project.berlios.de
4. Anneke Kleppe, Jos Warmer, The Object Constraint Language Second Edition, Getting Your Models Ready for MDA, 2003, Addison-Wesley
5. BERLIOS – Open Source Platform Berlios. http://www.berlios.de
6. The MathWorks Inc. MATLAB/Simulink. www.mathworks.com
7. MATLAB/Simulink. Examples in Documentation. Simulink Demos
8. IABG: Das V-Modell – Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell Kurzbeschreibung, 1997.
9. The MathWorks Automotive Advisory Board (MAAB) www.mathworks.com/industries/auto/maab.html
10. Controller Style Guidelines For Production Intent Using MATLAB, Simulink and Stateflow. www.mathworks.com/applications/controldesign
11. OMG: Meta Object Facility (MOF). Version 1.4. www.omg.org/technology/documents/formal/mof.htm
12. OMG: Model Driven Architecture (MDA). www.omg.org/docs/omg/03-06-01.pdf
13. NASA Flight Software Branch: C++ Coding Standard, 2003
14. HP: Code Advisor, http://www.hp.com/go/cadvise/