14 MATE - A Model Analysis and Transformation Environment for MATLAB Simulink

Elodie Legros[1], Wilhelm Schäfer[2], Andy Schürr[1], and Ingo Stürmer[3]

[1] Technische Universität Darmstadt, Real-Time Systems Lab
{legros,schuerr}@es.tu-darmstadt.de
[2] University of Paderdorn, Software Engineering Group
wilhelm@uni-paderborn.de
[3] Model Engineering Solutions, Berlin
stuermer@model-engineers.com

Abstract. In the automotive industry, the model driven development of software is generally based on the use of the tool MATLAB Simulink. Huge catalogues with hundreds of modeling guidelines have already been developed to increase the quality of models and ensure the safety and reliability of the generated code. In this paper, we present the MATLAB Simulink Analysis and Transformation Environment (MATE), a tool using metamodeling techniques and visual graph transformations to automate the analysis and correction of models according to these guidelines. The MATE approach is illustrated by a typical example, and compared to other classical approaches for model analysis.

14.1 Introduction

Nowadays, model-based development is common practice within a wide range of automotive embedded software development projects. In model-based development, de facto standard modeling and simulation tools such as MATLAB Simulink are used for specifying, designing, implementing, and checking the functionality of new controller functions. The quality and efficiency of the software are strongly dependent upon the quality of the model used for code generation. For that purpose, modeling guidelines such as the MathWorks Automotive Advisory Board (MAAB) guidelines [1] have been defined and are usually adopted. There are tools available to help the modeler check a model according to these guidelines, such as the Simulink Model Advisor (part of the Simulink toolbox), the Model Examiner [2] or MINT [3]. They assist the developer in reporting violations of block settings, model configurations, or modeling styles that do not comply with such guidelines. However, for huge controller models, this can add up to a few hundreds, or even a few thousands, violations that must be corrected manually by the modeler. That is a cumbersome, complex, and expensive task. An in-house case study at Daimler detected in a model more than 2,000

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 323–328, 2010.

guideline violations. After closer inspection, it was estimated that tools such as MATE could fix automatically up to 45% (900) of these 2,000 rule violations, and approximately 40% with user feedback. Only 8% required a manual correction. Finally, 4% remained undefined, i.e. the modeler had to determine whether the reported violations really infringed upon a modeling guideline (Cf. [4] for details).

For that reason we developed the MATLAB Simulink Analysis and Transformation Environment (MATE)[1]. This environment supports the detection of previously specified guideline violations. In addition to the Model Advisor or MINT, the Model Examiner and MATE also provide the possibilities of model repair operations. While the Model Examiner supports only static repair operations, i.e. without performing a model transformation (refactoring), MATE provides also layout improvements and modeling pattern instantiations. Apart from that, MATE is capable to perform high-level model analysis operations, such as control flow and data flow analysis.

14.2 Approach

Analysis as well as refactoring of MATLAB models demands full access to MATLAB's model repository. This requires an intimate knowledge of the API written in M-Script, a proprietary script language. Because both the used language and the tool's API evolved over many years, learning how to program reliable model checks and transformations using this approach costs time and efforts. It requires an intimate knowledge of the MATLAB API. MATE overcomes this problem by providing a layer of uniform API adapters on top of which visual graph queries can be developed in a more human friendly manner and on a considerably higher level of abstraction. Another benefit of the graph transformation is the possibility to express model transformations, and, hence, to define repair actions in an abstract way.

MATE provides two ways to analyze and repair models: *online* and *offline*. The corresponding architecture is represented in Fig. 14.1. The *online* modus enables an interactive analysis of a model within MATLAB. The communication between MATE and Simulink is realized through an Online-Adapter which supports all required read-and-write operations. The MATE-Tool represents the Java application controlling the execution of the analysis and transformation operations. On the other hand, the *offline* modus works on an efficiently processable main memory representation of the data representing the models to be checked and is generally used for complex analyses and corrections. The models are exported in their proprietary *mdl*-format, and imported in the Model Repository through the Reader/Writer as instances of the Simulink metamodel represented in Fig. 14.2. The module SDM Transformation - SDM Analysis defined in both parts (online and offline) of the MATE architecture represents the specifications of model analysis and correction.

[1] MATE project: *Model Engineering Solution* and *DaimlerChrysler* in partnership with the universities of Paderborn, Siegen, Kassel and the TU Darmstadt.

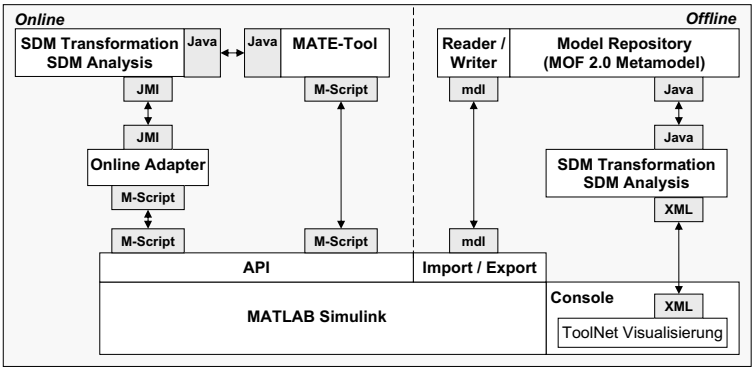

Fig. 14.1. MATE Architecture

Inside MATE, guideline specifications are based on a MOF 2.0 [5] compliant metamodel of Simulink. This metamodel is specified using the meta-CASE-Tool MOFLON presented in chapter 16. The real metamodel is very large and complicated. Therefore, only a very small version is presented in Fig. 14.2. The simplified metamodel consists of a single package Simulink containing only the most important classes. A Simulink model is a System that may contain a hierarchy of Subsystems with Blocks as leafs. Blocks are the atomic processing units. They are connected to each other by connecting their Outports and Inports via Lines.

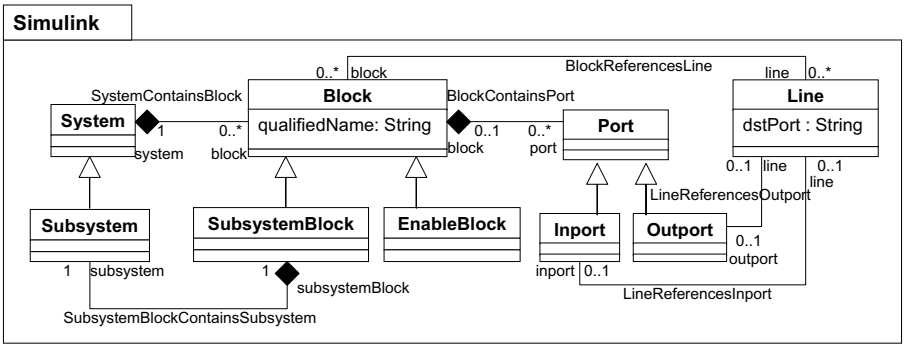

Fig. 14.2. Very simplified Simulink metamodel

This metamodel acts as graph schema for the specification of graph transformation rules. The technique of graph transformations is on the one hand applied for the detection of incorrect models as well as, on the other hand, for the (semi)automatic repair of identified errors. MATE uses the visual graph transformation approach of Story Driven Modeling (SDM) supported by the graph transformation tool Fujaba [6] and our plug-in MOFLON [7]. The next section gives an example of guideline specification with SDM. For more details about SDM, please refer to [8].

14.3 Application

MATE provides the analysis and correction of models according to modeling guidelines such as the MAAB guidelines [1]. One of the MAAB guideline concerns the naming of Enable Port blocks (jc_0281, "Naming of Trigger Port block and Enable Port block" [1]) The Enable block's name matches the name of the corresponding enable signal of the regarded subsystem (Cf. Fig. 14.3).

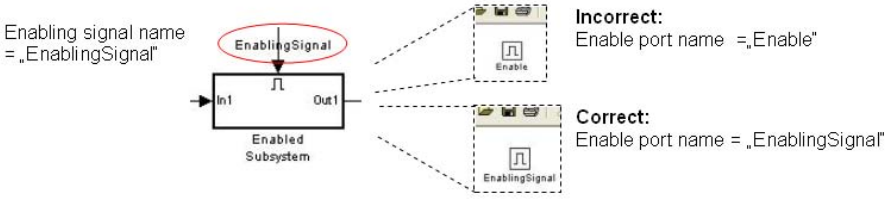

Fig. 14.3. Naming of Enable Port Block

Such a guideline can be implemented in different ways. It is state of the art that MATLAB modeling guidelines are implemented using the imperative scripting language M-Script. The M-Script specification of the guideline presented above would be the following:

```
function f_block_h = guideline_2(system, cmd_s)
  top_h = get_param(bdroot,'Handle');
  f_block_h = [];
  subsys = get_param(get_param(find_system(top_h, 'BlockType','EnablePort'), 'Parent'),
                                                                    'Handle');
  for k=1:length(subsys)
    subsys_handle = get_param(subsys{k},'Handle');
    porth = get_param(subsys{k},'PortHandles');
    enable_port_name = get_param(porth.Enable,'Name');
    enableh = find_system(subsys{k},'SearchDepth',1,'BlockType','EnablePort');
    enable_block_name = get_param(enableh,'Name');
    if ~(strcmp(enable_port_name, enable_block_name))
       f_block_h = [f_block_h;subsys_handle];
    end
  end % for
end % function
```

In fact, the implementation of model guidelines with M-Script is nothing else than traversing graph structures and implementing graph pattern matching operations with an imperative language. Thus, implementing guidelines with M-Script is rather a task of programming skills and detailed API knowledge than a task of a conceptual and well structured conversion of an informal description into a formal one.

Since modeling guidelines represent constraints on model elements or relations between model elements which have to be respected, the OMG's logic-based language OCL [9] can be used for a formal description of rules like the modeling guideline described above:

```
context SubsystemBlock inv:
if self.containedBlock ->exists(b:Block j b.oclIsTypeOf(EnableBlock))
```

```
then self.containingSubsystemBlock.incomingLine ->select( line j line.dstPort = "enable")
       ->collect(qualifiedName)
 -> intersection (self.containedBlock ->select(b:Block j b.oclIsTypeOf(EnableBlock))
       ->collect(qualifiedName))
 -> notEmpty()
endif
```

Though, OCL is not very well-suited for the specification of complex patterns, where we have to navigate along different paths through a model and to compare their results. Even worse, OCL is not able to define most modeling guidelines about naming conventions since the available operations on strings and characters are pretty poor. In the same way, guidelines requiring complex arithmetic operations cannot be defined using OCL.

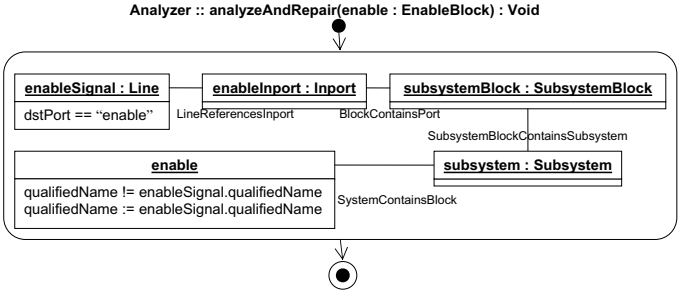

Fig. 14.4. Guideline specification with SDM

Therefore, we considered another approach when developing MATE, namely the visual graph transformation approach. To define these graph transformations, we use the visual SDM syntax. Fig. 14.4 simultaneously checks and fixes violations of the guideline presented above. It matches any occurrence of a pattern, where an EnableBlock and an EnableSignal object, which belong to the same Subsystem, do not have the same qualifiedName attribute. The line with the *":="*-operator inside the enable object rectangle assigns the name of the matched enableSignal to the regarded enable object.

This specification shows how visual graph transformations facilitate the search for more complex object/link patterns. Moreover, while OCL only defines model checking operations, the SDM graph transformations provide incorporated repair actions. In case of guidelines that cannot be translated into graph transformation (e.g. naming conventions), Java code can be embedded within a story diagram by using so-called *Java Statements*, whereas OCL proposes no alternative for guidelines that cannot be specified as OCL constraints. A more detailed comparison between the SDM approach and other classical approaches for the analysis and correction of models, as well as the M-Script and OCL specification of the guideline described forehand, can be found in [10].

14.4 Conclusion

The adoption of modeling guidelines for the design of automotive controller models is important. MATE supports the developer in analyzing MATLAB Simulink models and automatically or interactively transforming such models into guideline-compliant ones. This tool is based on the use of visual graph transformations to specify the guidelines. The advantages of graph transformation are the higher level of abstraction, and the possibility to define not only check but also repair actions. The used language SDM is a visual graph transformation language. The visual aspect facilitates the specification of patterns to be matched as well as the navigation through a model to find information to be checked, i.e. tasks which are frequently needed when specifying modeling guidelines.

Although the SDM graph transformations are pretty well-suited, they are not equipped for handling *all sorts* of modeling guidelines. Moreover, many guidelines require the definition of quite similar transformation rules. Therefore, we are just extending the SDM syntax with generic and reflective features to increase the reusability and expressiveness of story diagrams [11]. Since MATLAB Simulink is widely used in the automotive industry, MATE should not remain a pure research prototype but be developed according to the needs of the industry. Therefore, we are integrating MATE within the Model Examiner developed by *Model Engineering Solution* [2] so that it can be used and, hence, evaluated in the context of concrete industrial use cases.

References

[1] MathWorks Automotive Advisory Board Homepage, `http://www.mathworks.com/industries/auto/maab.html`
[2] Model Examiner Homepage, `http://www.model-engineers.com/our-products/model-examiner.html`
[3] Mint Homepage, `http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint`
[4] Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: Enhanced simulink and stateflow model transformation: The MATE approach. In: Proc. of MathWorks Automotive Conference (MAC 2007), June 19-20, Dearborn (MI), USA (2007)
[5] Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification. ptc/03-10-04 (2003)
[6] Fujaba Homepage, `http://www.fujaba.de`
[7] MOFLON Homepage, `http://www.moflon.org`
[8] Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn, Habilitation Thesis (2001)
[9] OCL Specification, `http://www.omg.org/docs/ptc/03-10-14.pdf`
[10] Amelunxen, C., Legros, E., Schürr, A.: Checking and Enforcement of Modeling Guidelines with Graph Transformations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Proceedings of the Third International Symposium on Applications of Graph Transformations with Industrial Relevance (October 2007)
[11] Amelunxen, C., Legros, E., Schürr, A.: Generic and Reflective Graph Transformations for the Checking and Enforcement of Modeling Guidelines. In: Proc. of the Visual Languages and Human-Centric Computing (VL/HCC 2008), pp. 211–218 (September 2008)