Tool Supported Quality Assessment and Improvement in MATLAB Simulink and Stateflow Models

Dietrich Travkin\textsuperscript{1}, Ingo Stürmer\textsuperscript{2}
\textsuperscript{1}University of Paderborn, travkin@uni-paderborn.de
\textsuperscript{2}Model Engineering Solutions, Berlin, stuermer@model-engineers.com

Abstract

Model-based development and automatic code generation are an established technology in the controller design process. Numerous modeling guidelines are built to improve the quality of the developed model and the generated code with respect to readability, robustness, safety, efficiency, maintainability and other aspects. Some guidelines are necessary to enable code generation. Manually checking guideline conformance of complex models as well as manually eliminating guideline violations can be error-prone and laborious.

In this paper we present an approach which provides developers with tool support for analysis and improvement of MATLAB Simulink and Stateflow models. Developers are aided in automatically detecting as well as fully-automatically or interactively eliminating guideline violations. In addition, instantiating modeling patterns is automated. Thus creating guideline compliant models is made easier. Furthermore, support in calculating model metrics is provided.

The automation of model analysis and metrics calculation aid developers with assessing the model quality while tool support for eliminating guideline violations and instantiating modeling patterns help to improve the model quality.

1 Introduction

Today, model-based development is common practice in a wide range of automotive embedded software development projects. In model-based development de facto standard modeling and simulation tools, such as MATLAB Simulink and Stateflow, are used for specifying, designing, implementing, and checking the functionality of new controller functions. Quality and efficiency of the software are strongly dependent upon the quality of the model used for code generation. Moreover, the model quality influences readability and thus the development and maintenance input. For this reason best practices are provided with the more commonly adopted modeling guidelines such as the MathWorks Automotive Advisory Board (MAAB) guidelines [18].

Commercial tools are available, such as the Simulink Model Advisor [17] or MINT [13], and research prototypes like MESA [4], which help the modeler to check a model with regard to guideline conformance. These tools assist the developer, for example, in reporting violations of block settings, model configurations, or modeling styles that do not comply with such guidelines.

In large-scale controller models, however, this can add up to several hundreds or even several thousands of violations that must all be manually corrected by the modeler. That is a cumbersome, complex, and ultimately expensive task.

A recent in-house case study at Daimler showed us that up to 80\% of guideline violations can be fixed fully or semi-automatically, i.e. involving some degree of user interaction (ref. [16] for details). For this reason, we developed the MATLAB Simulink and Stateflow Analysis and Transformation Environment (MATE), which is fully integrated into the MATLAB Simulink and Stateflow modeling environment. MATE provides support in detecting previously specified guideline violations, and in addition, (in contrast to Model Advisor, MINT, and MESA) enables to automatically apply predefined model transformations, for example repair operations, layout improvements, and modeling pattern instantiations. Furthermore, model metrics can be calculated, e.g. determining the system hierarchy depth or the number of all blocks in a Simulink model.
Given the functionality provided by MATE not only guideline compliance of Simulink and Stateflow models can be ensured. The tool also provides support to assess and improve the model quality. Detecting guideline violations reveals model parts with definitely low quality, while calculating metrics helps to identify model elements with abnormally high or low numeric properties (e.g. very high or low system hierarchy depth) and thus with potentially low quality. The model transformations help to increase the model quality by eliminating guideline violations, creating best practice modeling patterns, and improving the layout.

The MATE approach has already been briefly presented in [16, 15]. In this practice report paper we focus on the model transformation functionality, provide more insight into the techniques and contrast our work to other related research topics.

The remainder of this paper is structured as follows: In section 2 we motivate the usage of modeling guidelines and emphasize the necessity for tool support in the analysis and repair of models. After a short introduction of the MATE project in section 3 we describe typical model analysis and transformation operations provided by MATE in sections 4 and 5. The specification of these operations is depicted in section 6. Related work is addressed in section 7. Finally, conclusions are drawn in section 8.

2 Modeling Guidelines

An agreement to follow certain modeling guidelines is important to increase the comprehensibility (readability) of a model, facilitate maintenance, reusability, and extensibility, ease testing, and simplify the exchange of models between OEMs and suppliers. This is the purpose of MAAB guidelines and patterns [18]. Such publicly available guidelines are often supplemented by inhouse sets of modeling guidelines [10, 3], for example in order to check the models used for production code generation. The adoption of such guidelines can significantly improve the efficiency of generated code.

There are more than 80 MAAB guidelines. Daimler’s model-based development guidelines [10] consist of more than 200 modeling rules and patterns. Given this vast number of modeling rules, few reviewers are capable of keeping them all in mind1. Furthermore, since purely manual model reviewing and elimination of guideline violations is a laborious and error-prone process, it becomes expedient to check and repair models by means of automated tools.

1Experiences with model reviews are discussed in [14].

In a recent in-house case study at Daimler, we used the MINT framework to check a typical, large-scale controller model from the automotive domain. The emphasis of this case study was on (a) evaluating the maturity of the model with regard to production code generation and (b) checking the model in regard to guideline compliancy. It turned out that the model contained more than 2,000 guideline violations. Upon closer inspection, we estimated that up to 45 percent (900) of these 2,000 rule violations could have been fixed automatically, if tools such as MATE had been used [16]. Furthermore, we estimated that approximately 43 percent could have been repaired with user feedback (interactive repair operations). Only 8 percent actually required a manual correction. A final 4 percent remained undefined, which meant that the modeler had to determine whether the reported violation actually constituted an infringement of a modeling guideline.

3 MATE Project

Building more and more complex embedded systems, the automotive industry’s urgent need to increase the systems’ robustness, safety and efficiency led to a high number of modeling guidelines. Lacking a sophisticated tool support in assessing and increasing the model quality to enable the generation of safe and efficient controller code, e.g. by enforcing guideline conformance, the idea for the MATE2 project was born.

MATE is aimed to become a flexible, extendable development environment providing tool support in assessing and improving the quality of MATLAB Simulink and Stateflow models as well as their guideline conformant development. As a joint project of the universities of Darmstadt3, Kassel, Paderborn and Siegen as well as the companies Daimler and Model Engineering Solutions a MATE prototype is being developed since 2006.

Currently MATE supports the graph grammar based specification of modeling patterns (these can be best practice modeling patterns or patterns violating a certain guideline) and model transformations (e.g. operations eliminating previously detected guideline violations). Using these specifications the automated detection of modeling patterns and the execution of applicable model transformation operations are supported.

As an alternative to the graphical specification of complex operations it is possible to implement them

---

2MATE = MATLAB Simulink and Stateflow Analysis and Transformation Environment
3Technical University of Darmstadt
in Java or to re-use and apply already existing model analysis operations written in M-script. Using Java instead of the graphical specifications can be useful to implement complex algorithms, which are independent from the object-oriented model structures (e.g. complex mathematical operations), so that the benefit of a graphical specification is lost. But in both cases (Java and M-Script) a lower level of abstraction is used, which can significantly increase the complexity in cases of model structure description or traversal (cp. [1, 15]).

Based on the MATE capabilities stated above it is possible to

- detect guideline violations (structure, data flow, layout, properties, logical constraints,...)
- fully or semi-automatically eliminate guideline violations
- instantiate best practice modeling patterns
- apply modeling operations missing in MATLAB (e.g. layout operations, consistency checks,...)
- calculate model metrics (e.g. to assess model complexity)

MATE is fully integrated into MATLAB. Thus a MATLAB user can benefit from all MATE features at any time during or after development of a Simulink or Stateflow model.

4 Model Analysis

MATE provides support for performing several kinds of model analysis operations. These can detect guideline violating model structures, perform data flow analyses (cp. [16]), reveal elements with certain properties (e.g. having specific attribute values or satisfying some metric constraints), etc.

One of the guidelines used for code generation with Simulink/Stateflow diagrams and TargetLink (tl_0009) [3] requires product (and sum) blocks with fixed-point output data type not to exceed two input variables. If using more than two operands, scaling information for intermediate results, which is needed for proper fixed-point code generation, would be missing.

MATE provides the possibility to specify this kind of guideline violations as a modeling pattern. Given a pattern specification, MATE generates code which
is able to detect all occurrences of the specified pattern in a Simulink/Stateflow model. After specification of all (guideline violation) patterns to be found and generating the detection code, these operations are integrated into MATLAB’s Model Advisor framework [17]. Finally an engineer is able to detect all pattern occurrences in a given model. Together with a brief description of the detected problem these are listed in a result view (see figure 1). Furthermore the elements involved in the guideline violation can be highlighted in the diagram as illustrated in figure 2.

5 Model Transformations

MATE provides different kinds of operations that modify a MATLAB Simulink and Stateflow model. In this section, we discuss (1) model repair operations, (2) layout operations, and (3) design pattern instantiation.

5.1 Model Repair Operations

After detecting guideline violations during a model analysis phase, all applicable, predefined repair operations are suggested to eliminate the reported violations. Figure 1 shows the analysis results of a Simulink diagram using the report functionality of the Model Advisor. The report includes the suggested repair operation ”Cascade...” (bottom of the figure), which can be applied by clicking on the link. These repair operations and other operations are also applicable via context menu as shown in figure 1.

Product blocks with more than two operands, for example (see figures 2 and 1, highlighted block), can automatically be transformed into a cascade of product blocks with two operands (see figure 4). An engineer only has to specify the new product block’s output data type to add the missing scaling information for intermediate results. Since most guideline violations can be fully automatically fixed, or at least with few user interactions, such repair operations can reduce the input required to repair the model significantly.

5.2 Layout Operations

While modeling a system a developer has the option of applying "beautifying" model transformations via context menu. For example, it is possible to align Simulink blocks horizontally or vertically with each other, or to avoid line intersections by changing the port order (see figure 3: (a) the signal lines of the Product block are crossing; (b) the port order is changed and the lines are not crossing anymore). Such model transformations improve the arrangement of the model elements, thus easing model comprehension without significantly delaying the modeling process.

5.3 Design Pattern Instantiation

Guidelines like Daimler’s modeling guidelines [10] or the MAAB guidelines [18] usually contain best practice modeling patterns for recurring modeling tasks, for example patterns for if-then-else or switch-case constructs in Stateflow diagrams. The left hand side of figure 6 illustrates a typical pattern to be used when modeling control structures, such as the one specified by the pseudo-code on the right (figure 6).
The instantiation of modeling patterns during model design can be very time-consuming, for example when the nesting depth of the control structure increases. MATE speeds up this process and helps developers comply with guidelines by enabling modeling patterns to be instantiated from scratch with just a few clicks. Of course this possibility can also be used to improve the layout of existing diagrams by manually replacing non-compliant diagram structures with automatically instantiated patterns.

6 Specification of Analysis and Transformation Operations

MATE is a framework which provides some exemplary model analysis and transformation operations, but first and foremost offers support for specifying and applying new ones. Guideline violation patterns and repair operations are specified using graph transformation rules, in our case story diagrams (cf. [5, 12]). These rules are applied on the abstract syntax graph, which is a representation of the model to be analyzed.

The abstract syntax graph is an object structure. In this context each object represents an element in a Simulink or Stateflow diagram, i.e. in concrete syntax. The links between objects represent element relations, e.g. containment. Figure 8 illustrates a part of an abstract syntax graph representing the two product blocks and a connected constant block shown in figure 7. Each product block in the Simulink diagram is represented by an object of type Product, each signal line by a Line object, etc. Attributes name, type, value and others (not illustrated) contain additional information about the elements.

An abstract syntax graph is built based on an object-oriented meta-model for Simulink and State-
Figure 7. Concrete syntax example

Figure 8. Abstract syntax graph example

flow diagrams during the traversal of a given model, i.e. during its analysis. A small excerpt from this meta-model is shown in figure 9. It contains classes for most of the element types used in Simulink and Stateflow models as well as associations for their relations and attributes for their properties.

Although the model is incomplete (many block types are missing) the provided abstractions enable to traverse any Simulink model. The class Block for example can be used for any currently "unknown" block type. Otherwise the concrete classes like ChartBlock are used.

In order to build the meta-model we manually modeled all the classes and associations which we found important or necessary to realize our exemplary analysis and transformation operations. Afterwards we generated code for each of the classes.

One of the options provided by MATLAB to traverse or modify Simulink and Stateflow models is by running M-script commands. In our approach we manually adapted the generated accessor methods of our meta-model implementation to encapsulate M-script commands in the method bodies. Since the commands we used to access or modify a model do vary depending on the model element type (different command types for different model element types), we were not able to find a way to automatically select the right M-script command and to automate the implementation of accessor method bodies. This issue will be further explored in future, to ease the creation of a meta-model.

A big advantage of our approach is domain independence. The only required interface to enable modeling pattern detection and model transformation execution is the implementation of a meta-model for a domain’s abstract syntax tree, e.g. a meta-model implementation for Simulink and Stateflow models. The same approach has already been applied to detect design patterns [6] and so called anti patterns [2] in Java programms [12, 19, 11, 9].

6.1 Specifying Model Analyses

In our approach model analysis operations are specified using story diagrams [5]. These are special UML activity diagrams with nested collaboration diagrams, so called story patterns. A story pattern is a graph transformation rule which traverses or modifies the abstract syntax graph, i.e. it creates or removes objects, follows, adds, removes or reconnects links, or changes attribute values.

A simplified dialect of this language is used to specify modeling patterns. In contrast to story diagrams these so called pattern specification diagrams do not need any control flow. Only the structure to be found and its properties are specified.

An exemplary specification of a guideline violation pattern is illustrated in figure 10. The two rectangles represent objects to be found in an abstract syntax graph. In this case the diagram specifies that objects of type MathOperation and Outport which are

\[ \text{MathOperationWithTooManyInports} \]

\[ \text{create} \]

\[ \text{context} \]

\[ \text{outports} \]

\[ \text{block: MathOperation} \]

\[ \text{numberOfInports > 2} \]

\[ \text{dataType = "(?i)(int8|uint8|int16|uint16|int32|uint32)"} \]

Figure 10. Guideline Violation Pattern

4In future this meta-model will be extended.
connected via an \textit{outports} link and which satisfy the given attribute constraints are to be found.

Since \textit{MathOperation} is a super type of the meta-model classes \textit{Product} and \textit{Sum} (cp. meta-model in figure 9) either a product block or a sum block with more than two input ports is represented by the object named \textit{block}. The regular expression specified for the \textit{outport} object constrains the block to have a fixed-point output data type. Altogether this diagram specifies situations as illustrated in figure 2.

Given a specification of a modeling pattern as illustrated in figure 10 MATE is able to generate Java code, which represents a search for the specified object structure and can be executed from within MATLAB’s Model Advisor framework. For more details on the specification and execution of analysis operations please refer to [16, 1, 12, 19, 11].

### 6.2 Specifying Model Transformations

A model transformation operation in MATE is specified using a story diagram [5]. Figure 11 illustrates an exemplary specification of a model transformation.

Given a \textit{Product} object, i.e. a product block, as a starting point, this transformation tries to find the specified object structure in the abstract syntax graph, then it removes all the elements marked with ”destroy” and adds all the elements marked with ”create”. In this case the transformation replaces product blocks with more than two operands by a cascade of product blocks with only two operands. Given such a graphic specification of a transformation, i.e. a story diagram, MATE offers the option of generating Java code that executes the specified operation. The code can be nested into the Model Advisor framework so that the specified transformation can be applied to Simulink and Stateflow models.

### 7 Related Work

Well-known examples of other MATLAB Simulink and Stateflow analysis tools are MathWork’s Model Advisor [17] and MINT [13]. Both rely on the execution of MATLAB M-scripts to identify modeling rule violations. Other tools like MESA [4] enable the specification and verification of OCL constraints. However, none of these tools provide support for repairing a model, improving its layout, and instantiating design patterns. MATE, in contrast, in addition to the functionality of specifying guideline violation patterns, which can be automatically detected in a model (see [16] for more details), supports model transformations.

Furthermore, the implementation of guideline checks in M-script is a great deal more complex than implementing them with the MATE approach. With MATE, the rules are specified on a higher level of abstraction using graph transformation rules. The rule implementation is generated from this high-level specification. Moreover, MATE uses an object-oriented model of Simulink and Stateflow diagrams, which acts as a further layer of abstraction. It hides the Simulink
and Stateflow API operations (M-script) behind a uniform, consistent interface for querying and modifying all model elements.

The MESA project [4] also focuses on the application of a high-level analysis specification language with code generation support. However MESA does not rely on graph and rule-based specification techniques like MATE. Instead it uses the textual and logic-based Object Constraint Language (OCL) of OMG. This makes descriptions of object structures not as easy to understand as the graph transformation rules used by MATE.

Different specification techniques, especially M-scripts, OCL and graph grammar based specifications, have been compared in [1]. In some cases graph transformation rules may also become complex, for example when specifying complex layout algorithms. But nevertheless, in general, the graph grammar based approach offers better support for the realization of analyses and transformations of graph-based or object-oriented models.

A comparable approach to MATE is aquintos’ PREEvision tool suite, which also uses graphic specifications for model analysis and transformations. However in contrast to MATE, PREEvision is aimed towards model-to-model transformation techniques (e.g. transformation of a Simulink and Stateflow model into an ASCET/SD model) instead of model modifications. PREEvision does not focus directly on the analysis of modeling guideline violations and does not support model repair functionalities with user feedback or pattern instantiation.

There are further research approaches that support model analysis and transformation [7, 8], but these focus on UML and Statechart diagrams.

8 Conclusion

Adopting modeling guidelines for the design of automotive controller models is vital. The huge number of modeling rules and patterns that the developer must keep in mind for tool support in order to create, check, and repair controller models with regard to guideline compliancy. Furthermore, the equally large number of violated guidelines makes model reworking cumbersome and prone to error.

MATE supports developers in analyzing MATLAB Simulink and Stateflow models and automatically or interactively transforming such models into guideline-compliant ones. In addition, MATE provides high-level editor functions, which help developers to design guideline-compliant models by generating predefined and configurable modeling patterns or using layout operations.

Specifying analysis and transformation operations using graph transformation rules has the advantage to use a high level of abstraction. In contrast to MATE other tools providing analysis support for MATLAB Simulink and Stateflow models are using imperative programming languages like M-script [17, 13] or logic-based languages like OCL\textsuperscript{5} [4] to specify analysis operations or guideline violations to be found. By using model-based specification techniques and automated code generation in our MATE approach implementation details are no longer relevant when specifying analysis and transformation operations.

References


\textsuperscript{5}OMG’s Object Constraint Language


