

# The MATE Approach: Enhanced Simulink<sup>®</sup> and Stateflow<sup>®</sup> Model Transformation

## Ingo Stürmer

Model Engineering Solutions, Berlin, Germany

## Ingo Kreuz

DaimlerChrysler AG, Research and Technology, Sindelfingen, Germany

## Wilhelm Schäfer

University of Paderborn, Department of Computer Science, Paderborn, Germany

## Andy Schür

Technical University of Darmstadt, Real-Time Systems Lab, Darmstadt, Germany

## ABSTRACT

In this paper we present the Model Advisor Transformation Extension (MATE). The purpose of MATE is to complement the functionality of the MathWorks MATLAB<sup>®</sup>, Simulink<sup>®</sup>, and Stateflow<sup>®</sup>, Model Advisor, and to extend the tool's capabilities with regard to model transformation and improvement functions. Examples of MATE features are: automatic or interactive model analysis and repair functions; design pattern instantiation; beautifier operations. We present typical use cases for MATE and discuss the relevance of the MATE approach compared with other available tools and approaches.

## 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, and Stateflow are used for specifying, designing, implementing, and checking the functionality of new control functions. The controller software can be either 1) manually developed by programmers on the basis of the controller model or 2) generated automatically by a code generator. Either way, the quality and efficiency of the software are strongly dependent upon the quality of the model. Code efficiency is important due to the limited resources of the embedded system that will run the generated code. Therefore, generally accepted modeling guidelines--such as the MathWorks Automotive Advisory Board (MAAB) guidelines--are usually adopted. They support the developer in preventing typical modeling problems. The MathWorks Model Advisor assists 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 hundred -- or even a few thousand -- violations that must be corrected manually by the modeler. That is a cumbersome, complex, and expensive task.

A recent in-house case study at DaimlerChrysler showed us that automated and partly interactive model corrections (i.e., transformations) can reduce the effort of model rework by up to 70 percent. This represents a significant improvement over doing automated model checks with manual model rework. This finding led us to develop a Model Advisor Transformation Extension (MATE) that extends the capabilities of the MATLAB, Simulink, and Stateflow Model Advisor. The main purpose of the MATE add-on is to complement the analysis functionality of the MathWorks Model Advisor and to shift the focus from analytical to constructive model quality improvement. It offers four different categories of model transformation and improvement functions:

1. **Automatic repair functions.** These can potentially eliminate up to 60 percent of MAAB guideline violations by means of automated model transformations. For example, a typical, straightforward repair function would be to set the actual view of every subsystem view to 100 percent.
2. **Interactive repair functions.** These are high-level analysis and repair functions that require user-feedback, since the required information cannot be determined automatically. One example would be autoscaling support functions, which check and calculate scaling errors by means of data flow analysis and generate proposals for

how to eliminate these errors. Such functions often need additional fixed-point scaling information or input about handling specific data types.

3. **Design pattern instantiation.** The MAAB guidelines provide a rich set of modeling patterns, such as the Simulink *if-then-else-if* pattern, and flowchart *switch-case* constructs. These pre-defined, MAAB guideline-conformant pattern instances can be automatically generated from a pattern library. For example, MATE supports instantiation of a parameterized flowchart *switch-case* pattern, the nesting deepness of which can be configured by the modeler.
4. **Model “beautifying” operations.** Providing an appropriate model layout is important and often required. For example, it is common practice to locate all Simulink in-ports on the left side of a subsystem. MATE provides a set of beautifying operations that, for example, align a selected set of blocks to the left, right, or in the middle relative to a reference block. Furthermore, signals leading into ports of a subsystem can be easily interchanged when signal lines are overlapping.

These MATE features focus on the following goals:

1. Provide embedded tool support for the model-based development of embedded software on the basis of MATLAB, Simulink, and Stateflow models.
2. Support the developer with regard to batch-oriented guidelines conformance and rule-checking.
3. Provide automated and partly interactive model rework to achieve guideline conformance.

In the following sections, we present fundamentals of the MATE tools as well as typical application scenarios for MATE.

## MODELING GUIDELINES AND THE APPLICATION OF MODEL ANALYSIS TOOLS IN PRAXIS

An agreement to follow certain modeling guidelines is important to increase the comprehensibility (readability) of the model, facilitate maintenance, ease testing, reuse, and extensibility, and simplify the exchange of models among OEMs and suppliers. This is the purpose of the MAAB guidelines and patterns [4]. Following the modeling conventions stated in those guidelines supports the translation of the model into proper code. Such publicly available guidelines are often supplemented by in-house sets of modeling guidelines in order to check the models used for production code generation. The Model-based Development Guidelines of DaimlerChrysler consists of more than 200 rules and patterns. The guidelines are published via the e-Guidelines Server [5]; a Web-based infrastructure for publishing and centrally administering different sets of guidelines. The DaimlerChrysler guidelines are structured into basic categories such as:

- Fundamental modeling rules (naming conventions, model structuring)
- Tool-specific modeling patterns and guidelines (Simulink, Stateflow, mixed Simulink and Stateflow)
- Autocode intend guidelines
- Model-based testing guidelines
- Safety-pattern definitions
- Guidelines for variant modeling.

The huge number of modeling rules necessitates that models be checked for guideline compliancy by means of automated tools, since pure manual model reviews are laborious and prone to error. Furthermore, few reviewers would be capable of keeping all the modeling rules to be checked in mind. However, experience has shown that a combined model and code review can significantly improve the quality of the model used for code generation [6].

Tools are also available to help the modeler check a model with regard to guideline conformance, such as the Simulink Model Advisor [1] and MINT [2]. Both tools provide a framework for executing M-scripts, which basically check for guideline conformance. In a recent in-house case study, we used the MINT framework in order to check a typical but huge controller model from the automotive domain. The emphasis of this case study was to (a) evaluate the maturity of the model with regard to production code generation and (b) to check the model concerning guideline compliancy. It turned out that the model contained more than 2,000 guideline violations. After 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

(see Figure 1). Furthermore, we estimated that approximately 40 percent could have been repaired with user feedback (interactive repair functions). 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 violations really infringed upon a modeling guideline

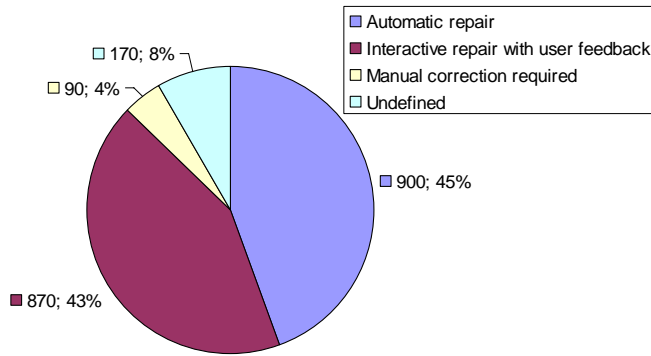


Figure 1. Case study: Overview of modeling guideline violations that can be repaired automatically using MATE (estimation).

To sum up: Modeling guidelines are accepted measures for improving quality when using a model-based development approach. A manual model review with regard to guideline conformance is time-consuming and prone to error, due to the huge amount of guidelines to be checked. The burden of correcting rule violations manually cannot be prevented, even when using a tool that analyzes a model for guideline conformance automatically. But high-level analysis functions, in combination with automatic repair functionality, can significantly reduce the effort required for model rework. Such high-level analysis and repair functions are described next.

## MODEL ANALYSIS AND TRANSFORMATION

In the following sections we present an example that shows how high-level analysis functions of MATE can be adopted in order to improve the quality of a model used for code generation.

### DATA FLOW ANALYSIS OF FIXED-POINT COMPUTATION

Fixed-point arithmetic is the preferred calculation method in embedded software development for reasons of limited resources and code efficiency. Fixed-point numbers allow us to express a real number as an integer by specifying its word size and the location of the binary point (power-of-two scaling). The main drawback of fixed-point numbers is their limited precision when representing physical values. For example, when a long numeric format is converted into a shorter one, a numerical error, termed a *quantization error*, is introduced. In order to keep quantization errors as low as possible, scaling is used to maintain precision. Fixed-point representation with scaling information can be individually configured for every arithmetical Simulink block. The dialog for specifying the fixed-point data type and the respective scaling information for a product block is shown in Figure 2 (upper part). Here it is also possible to define the rounding method, which rounds the integer calculation toward *zero*, *nearest*, or *floor*. In the following example, we will examine the quantization error for the *nearest* rounding method.

The rounding errors of a product block ( $z=x*y$ ;  $x=input$ ;  $y=input$ ;  $z=output$ ) and the intervals of the input and output values can be calculated with the following formulas [8]:

Input interval:  $\Pi_x = [x_{min}, x_{max}]$ ,  $\Pi_y = [y_{min}, y_{max}]$ , Scaling factors of input and output:  $S_x, S_y, S_z$

Input error:  $\Delta_x, \Delta_y$ , Rounding error:  $\begin{cases} \leq S_z / 2, & \text{if } S_z > (S_x \cdot S_y) \\ 0, & \text{otherwise} \end{cases}$

Output error:  $\Delta_z \leq \Delta_x \cdot \|\Pi_y\|_\infty + \Delta_y \cdot \|\Pi_x\|_\infty + \Delta_x \cdot \Delta_y + \Delta_{prod}$

$$\text{Output interval: } \left[ \left( \prod_x \cdot \prod_y \right) \cdot \min - \Delta_{prod}, \left( \prod_x \cdot \prod_y \right) \cdot \max + \Delta_{prod} \right]$$

MATE implements a data flow analysis for interval and output error calculation, which helps the modeler to find imprecision in the fixed-point arithmetic. The model in Figure 2 represents the fixed-point calculation of the formula:  $y = x^3 + x^2 + x$ .

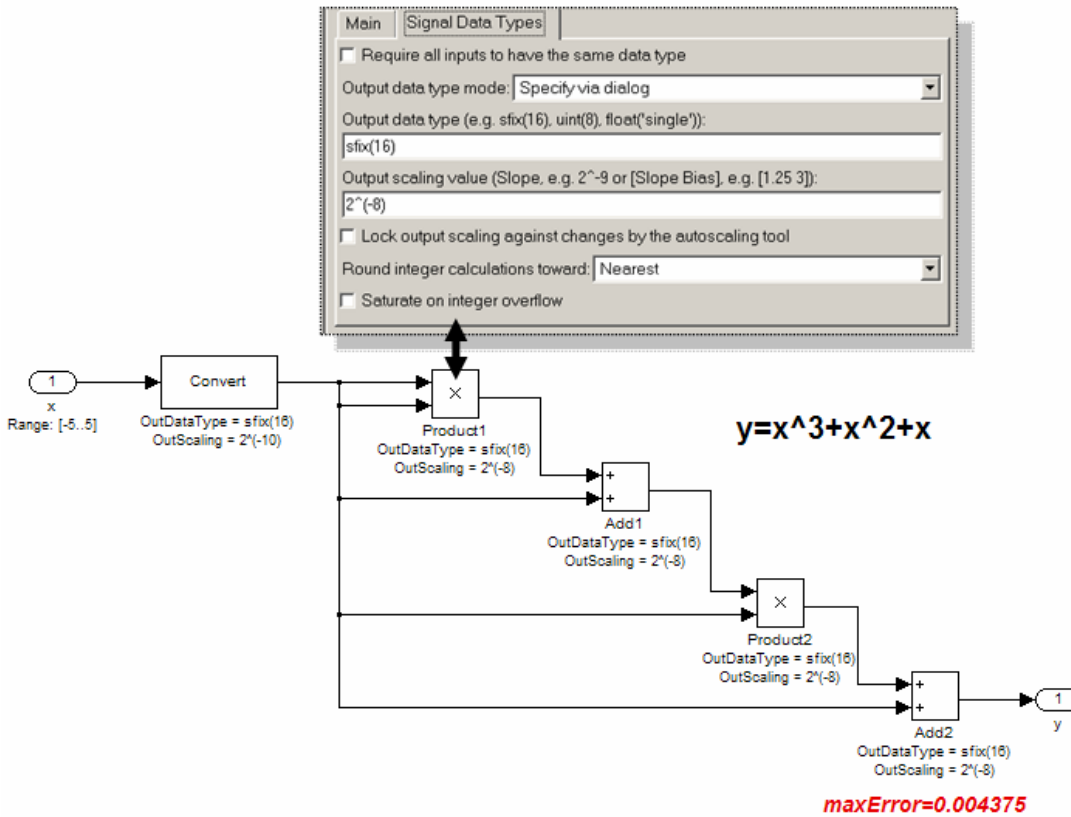


Figure 2. Example of using MATE for error calculation in fixed-point arithmetic (maxError).

A modeling guideline can require, for example, that the maximal scaling error (*maxError*) is not bigger than 0.5 percent of the possible output values  $y$  for this calculation. The propagation of the scaling errors, starting from the input  $x$  with values in the range between  $[-5..5]$  is  $\text{maxError}=0.004375$  for  $x=0.5$ . This is bigger than 0.5 percent of the possible output values. The propagation of scaling errors, for example with  $x=-2$ , results in the acceptable  $\text{maxError}$  of 0.03.

The implementation of the data flow analysis was realized with the open-source project *Fujaba* [7,11] and the metamodel plugin *MOFLON* [9]. *Fujaba/MOFLON* allows for the visual specification of graph pattern matching on a high abstraction level, while supporting the OMG-Metamodeling standard for model-based software development (MOF 2.0) [10]. We present a graph pattern search and model transformation algorithm with *Fujaba/MOFLON* in the following example.

### MODEL TRANSFORMATION FOR FIXED-POINT ARITHMETICS

In the previous example, we saw how MATE implements high-level analysis checks, based on data flow analysis. Here, we present model analysis with model transformation.

Arithmetical operations, such as a multiplication with two or more operands, are carried out in Simulink with product blocks. A product block with three inputs is shown in Figure 3 (left). This way of modeling might be problematic when the model is translated into fixed-point code by means of a code generator. Intermediate results have to be calculated, which cannot be determined by the code generator automatically. For this reason, product blocks with more than two inputs are forbidden for production code generation in one of the guideline collections used. MATE can be used for detecting all product blocks with more than two inputs. Every block it finds can be transformed into a cascade of product blocks with more than two inputs. The missing scaling information is supplemented via a user dialog (not shown in Figure 3).

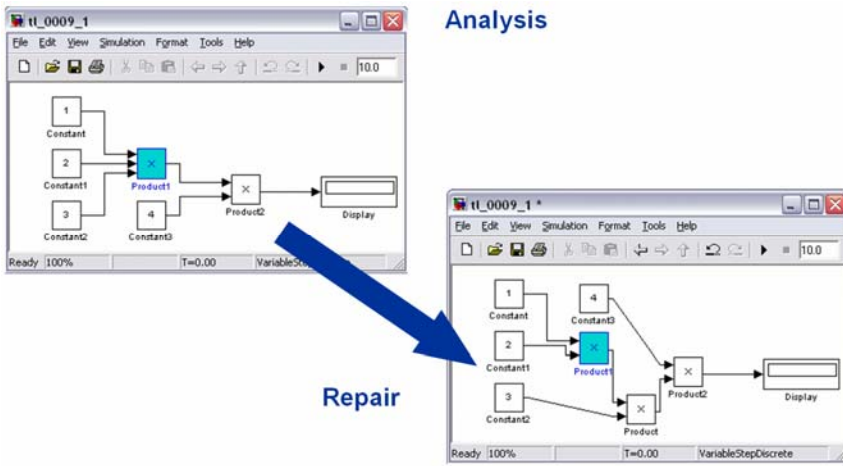


Figure 3. Example for MATE analysis and repair function: Product blocks with more than two inputs.

Two different rules are provided in Figures 4 (a) and (b), which are used for checking the correct use of product blocks. Figure 4 (a) uses the metamodel attribute `numberOfInPorts` in order to find product blocks (`p:Product`) with more than two inputs. Furthermore, it checks whether the out-port of the product block is an integer data type (for simplicity reasons, the Simulink fixed-point data types are not shown). If such a product block is found within the model, it is highlighted. The second pattern in Figure 4 (b) searches for product blocks with a vector as an input. If the port width of the in-port is greater than two (i.e., more than two elements are within this vector), the block is also marked as erroneous. In order to ensure that the first rule in Figures 4 (a) does not also match vectors or busses, the crossed-out node with "`portWidth > 1`" is used (negative application condition). Both patterns are summarized in the general pattern `ProductWithTooManyOperands` (Figure 4 (c)).

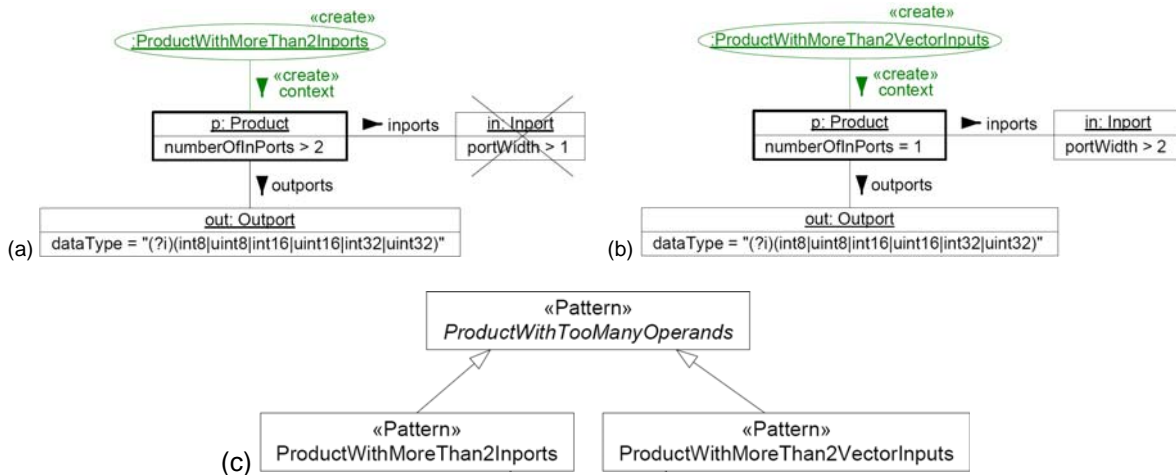


Figure 4. MATE analysis rule for product blocks with more than two inputs.

On the basis of the analysis rules shown above, unidirectional transformation rules can be used in the form of story diagrams in order to realize a model transformation. These transformation rules can be used to repair the highlighted model patterns (e.g., product blocks with more than two inputs), and to transform the model into a guideline-compliant pattern. The transformation rule shown in Figure 5 also uses the model analysis rules from Figure 4. The story diagram

allows for specifying model elements to be deleted («destroy») or created («create»). A loop for this atomic rule defines that it be applied until no other pattern match can be found (i.e., a violation of the regarded guideline).

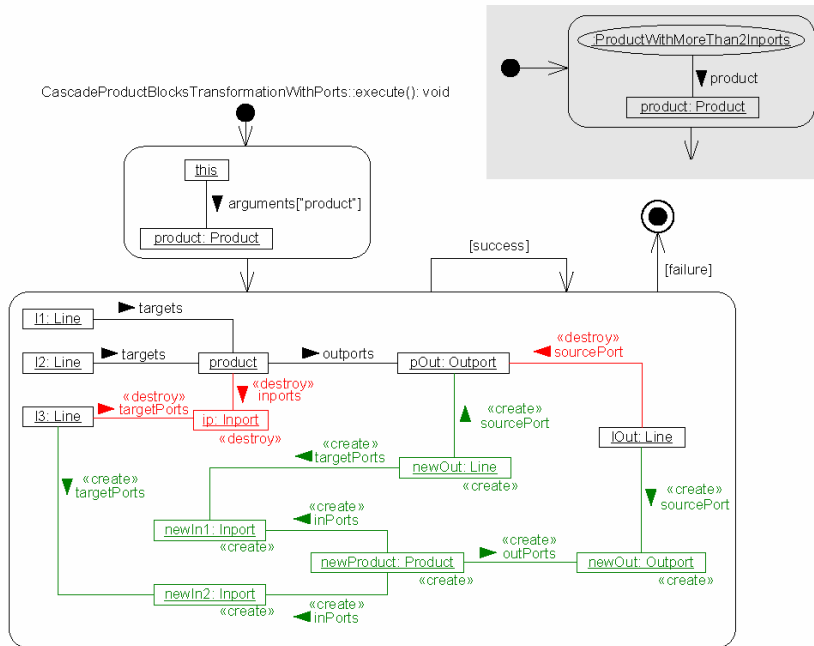


Figure 5. MATE transformation rule for product blocks with more than two inputs.

The result of the transformed model pattern in Figure 3 (left) is shown in Figure 3 (right). The integration of the transformation into MathWorks Model Advisor is shown in Figure 6.

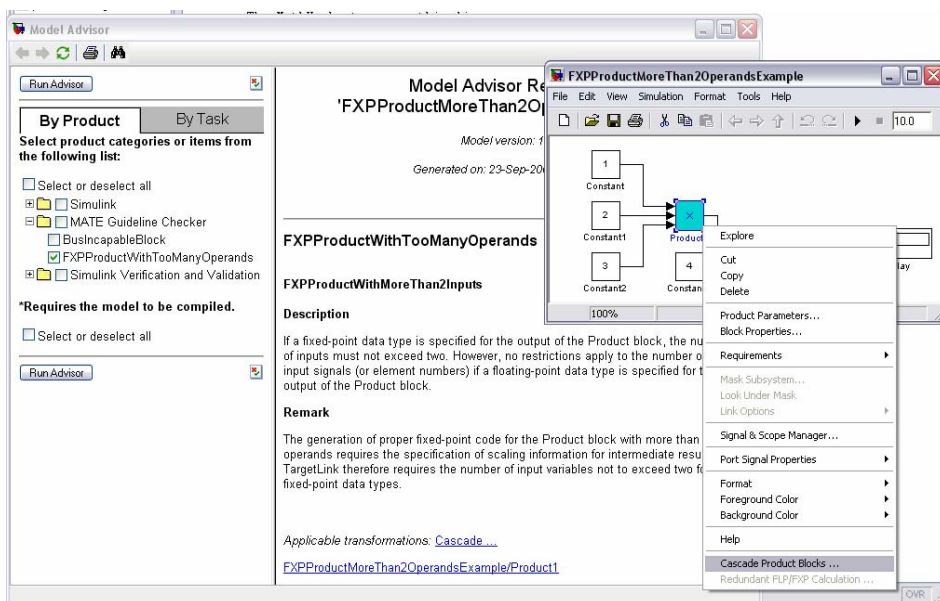


Figure 6. MATE analysis and transformation integrated into MathWorks Model Advisor.

## MATE ARCHITECTURE

The MATE project is based on the CASE tool Fujaba [7, 11], which supports multiple UML diagrams as well as Java code generation. In particular, it is possible to graphically describe graph transformations and to generate code that realizes the transformations. Fujaba and MATE are run in the Java runtime environment provided by MATLAB (see Figure 7).

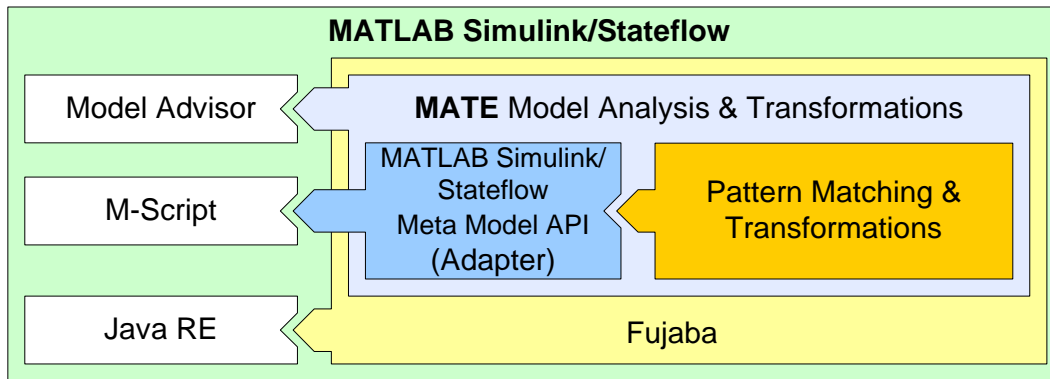


Figure 7. MATE architecture

The MATE environment is integrated into the MathWorks Model Advisor framework. The user interacts with MATE through the Model Advisor dialog to select the guidelines to be checked, start the analysis of a Simulink or Stateflow model, and display the analysis results. In addition to the analysis results, the possible model transformations (repair functions) are suggested in the results view. Execution of automatic and interactive repair functions can be triggered by selecting them in the generated Model Advisor results view.

To enable the analysis of Simulink and Stateflow models, MATE uses a Pattern Matching & Transformations component. Given a metamodel for MATLAB, Simulink, and Stateflow diagrams, this component provides the capability to match the graph-based description of guideline violations (patterns) and to perform graph transformations (repair functions) on a model. The Java code of this component is generated from high-level story diagrams, using the Fujaba graph transformation environment's compiler.

The metamodel used to describe guideline violations and repair functions is implemented in Java, too. Its interface was generated from a MATLAB, Simulink, and Stateflow metamodel, so that it satisfies the requirements of the Pattern Matching & Transformations component. To enable reading and modification of MATLAB, Simulink, and Stateflow models, the metamodel API acts as an adapter to the MATLAB environment. Therefore, the metamodel implementation internally uses M-script statements.

## 5. RELATED WORK

Well-known examples of other MATLAB, Simulink, and Stateflow analysis tools are MathWork's Model Advisor [1] and MINT [2]. Both rely on the execution of MATLAB M-scripts (i.e., checks) to identify modeling rule violations within Simulink and Stateflow models. The implementation of such checks is considerably more complex than implementing them with the MATE approach. With MATE, the rules are specified on a considerably higher level of abstraction, using a combination of visual model transformation/analysis rules and UML activity diagrams controlling the application of the visual rules. The Java implementation of the rules is generated from this high-level specification. Apart from that, MATE uses a Simulink and Stateflow metamodel, which makes it possible to specify analysis as well as transformation rules in a compact and simplified way. This metamodel acts as another layer of abstraction. It hides the operations of the MATLAB, Simulink, and Stateflow M-script API behind a uniform interface that uses the same "programming style" for creating and querying all modeling elements.

THE MESA project [3] also focuses on the application of a high-level analysis specification language with code generation support. But MESA does not rely on visual and rule-based specification techniques like MATE; rather, it uses the textual and logic-based Object Constraint Language (OCL) of OMG. A comprehensive comparison between FUJABA story diagrams and OCL constraints with regard to expressiveness, implementation effort, readability, and industry acceptance has not yet been done.

Table 1 summarizes and compares the main features of the four above-mentioned MATLAB, Simulink, and Stateflow analysis and transformation frameworks, starting with an emphasis on their functionalities from a user's point of view. All

four frameworks support the definition and execution of guideline checks as well as the generation of reports that list violated constraints (plus suggested repair operations in the case of MATE). Furthermore, all frameworks except MINT also allow for the definition of metrics that do not simply return a Boolean value, but use enumerations or numbers to indicate the quality of analyzed models. MATE is the only tool that offers functions for the semi-automatic manipulation of models. Of course, these functions could also be implemented using M-scripts and added to the Model Advisor or MINT framework. However, based on our experiences with implementing guideline checks with M-scripts on one hand and the visual story diagram specification language of MATE on the other hand, we can safely state that the implementation of model transformations using M-scripts would certainly require considerably more effort (by a factor of 2 to 3) and result in definitely less readable and maintainable code. Although the MESA project is planning to define model transformations, that is still out-of-scope because OCL supports only the specification of analysis operations on models.

Table 1: Tools for model analysis: comparison of features.

Feature	MA	MINT	MESA	MATE
Guideline checking	+	+	+	+
Model metrics	+	+	+-	+
Report generation	+	+	+	+
Automatic repair functions	-	-	-	+
Interactive repair functions	-	-	-	+
Enhanced editing functions	-	-	-	+
Model Advisor Integration	+	-	-	+
XML export/import	-	-	+	(+)
Model repository	-	-	+	(+)
Object-Oriented Standard	-	-	+	+

From an implementation point of view, the four frameworks adhere to different integration strategies. Both the Model Advisor and MINT analyze MATLAB, Simulink, and Stateflow models directly via M-script, whereas MESA exports models to a separate repository with an object-oriented API, using XML export/import functionality. MATE also supports export to, and import from, a separate model repository, but puts a main emphasis on the deep integration of analysis and repair functions into the MATLAB tool suite and its Model Advisor add-on. The essential difference between the pure Model Advisor approach and MATE concerns – except for the additional model transformation functions – is the existence of a Java API implemented on top of the available M-Script API. This API considerably simplifies the manipulation of Simulink and Stateflow models coded manually, but it is usually used by code that has been generated from high-level Fujaba SDM diagrams.

## CONCLUSIONS

In this paper, we presented the Model Advisor Transformation Extension (MATE). As far as we know, it is the first and only tool that offers integrated support for both model review *and* refactoring activities. MATE extends the MATLAB, Simulink, and Stateflow environment and complements the Model Advisor's analysis functionality with respect to batch-oriented and interactive model transformation and improvement functions. We presented examples of high-level analysis and repair functions and gave the reader an impression of the tool's architecture and its visual specification language. Compared to other available tools, MATE is so far the only one capable of performing model transformations that 1) eliminate guideline violations, 2) instantiate recommended design patterns, and 3) perform needed layout adjustments or model refactoring operations. From a user's point of view, MATE is not a separate tool with its own interface, but an add-on that is smoothly integrated into the Model Advisor framework. From the tool developer's point of view, MATE has the advantage of allowing guideline checks, repair actions, and so forth to be developed on different levels of abstraction, either in Java or in a visual rule-based model transformation language. It uses the graph transformation tool Fujaba, with a UML-like syntax and MOFLON plug-in that offers OMG standard-compliant metamodeling and model transformation functionality.

## REFERENCES

1. The MathWorks: <http://www.mathworks.com/products> (2006).
2. Ricardo, Inc., MINT: <http://www.ricardo.com/mint> (2006).
3. Farkas T.; Hein, Ch.; Ritter, T.: "Automatic Evaluation of Modelling Rules and Design," in *Second Workshop: From code centric to model centric software engineering: Practices, Implications and ROI*. Bilbao, Spain (2006).
4. Mathworks Automotive Advisory Board (MAAB): "MAAB Controller Style Guidelines for Production Intent." Release V2.0, The Mathworks, Inc., 2006.
5. Model Engineering Solutions: <http://www.e-guidelines.de> , <http://www.model-engineers.com/products.htm> (2007).
6. Stürmer, I.; Conrad, M.; Fey, I.; Dörr, H.: "Experiences with Model and Autocode Reviews in Model-based Software Development." *Proceedings of Third International ICSE Workshop on Software Engineering for Automotive Systems (SEAS'06)*, pp. 45-51, 2006.
7. Fujaba: <http://www.fujaba.de> and <http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html>
8. Sohn, M., "Correctness-terms for model-based code generators" (in German). Diploma thesis, University of Halle-Wittenberg, Department of Computer Science, Jun. 2006.
9. Technical University of Darmstadt, MOFLON: <http://www.moflon.org>
10. Amelunxen, C.; Königs, A.; Rötschke, T.; Schürr, T.: "MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations", in: A. Rensink, J. Warmer (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*, Heidelberg: Springer Verlag, 2006; *Lecture Notes in Computer Science (LNCS)*, Vol. 4066, Springer Verlag, 361--375.
11. Nickel, U.; Niere, J.; Zündorf, A.: Tool demonstration: The FUJABA environment, in *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, ACM Press (2000), 742-745

## CONTACTS

Ingo Stürmer: [stuermer@model-engineers.com](mailto:stuermer@model-engineers.com)

Ingo Kreuz: [ingo.kreuz@daimlerchrysler.com](mailto:ingo.kreuz@daimlerchrysler.com)

Wilhelm Schäfer: [wilhelm@upb.de](mailto:wilhelm@upb.de)

Andy Schürr: [andy.schuerr@es.tu-darmstadt.de](mailto:andy.schuerr@es.tu-darmstadt.de)