A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink

Holger Giese, Matthias Meyer, Robert Wagner
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Strae 100
33098 Paderborn, Germany
[hg|mm|wagner]@uni-paderborn.de

ABSTRACT

Nowadays, the software for electronic control units in embedded systems is often developed using a chain of model-based development tools. For example, in the field of automotive systems often tools like Matlab/Simulink and related code generation tools such as dSPACE TargetLink are used. To ensure that the produced models can be processed as expected by the employed tool chains, i.e., fulfill all requirements of the later process phases, they have to adhere to a large number of often tool specific as well as organization specific guidelines. In this paper, we present a first prototype for a tool which supports the specification and checking of guidelines for the Matlab/Simulink environment. In addition, we discuss first ideas for the specification of guidelines using examples in Matlab/Simulink notation as well as the semi-automatic correction of guideline violations.

1. INTRODUCTION

Today, a model-based approach is often employed to develop the software for electronic control units (ECUs) in embedded systems such as automotive systems. Tools such as Matlab/Simulink and related code generation tools such as dSPACE Target Link permit the developer to work most of the time only at a higher level of abstraction with the models rather than the code.

To ensure that the model-based development works in practice, it is not sufficient to only use the existing tools with all their features. Instead, the employed modeling features and even the models have to be restricted to ensure that the produced models can be processed as expected across the different tools of the employed tool chain. In addition, the different models employed throughout the process should fulfill specific requirements to ensure that they are appropriate for the next process phase.

To cope with these additional constraints on the employed models in each specific process phase, the original equipment manufacturers (OEMs), suppliers, as well as tool vendors started to develop their own guidelines which capture these constraints (cf. [1]). As most of them come up with rather large catalogs of hundreds of guidelines, the developers require tool support to cope with these guideline catalogs.

First approaches for the automatic guideline checking are for example MINT [7] and the Matlab Model Advisor framework [4]. These approaches are based on the M-Script language. However, solutions based on M-Scripts or other programmed rules have severe disadvantages. The programming and maintenance of the rules is very expensive and requires detailed knowledge of the Matlab/Simulink model. Therefore, a higher abstraction description of the guidelines which is feasible also for the domain experts is more attractive.

To provide such a higher level solution, we present in this paper the required concepts and a first prototype for checking guidelines in the Matlab/Simulink environment. Based on the graph transformation capabilities of the Fujaba Tool Suite, guidelines can be specified employing graphical though formal rules which refer to a clear adapter metamodel. These specifications can be executed to detect guideline violations automatically. To ease the specification of the guidelines and make it applicable also for domain experts, the prototype further supports to derive the guideline rules from examples given within the Matlab/Simulink tool. In addition, we sketch which extensions to our prototype are required to support the automatic correction of guideline violations.

2. EXAMPLE GUIDELINE

In the automotive field, high quantities and cost pressure often require control algorithms to be implemented in fixed-point code [1]. In practice, however, a controller model is developed in floating-point arithmetic initially and transformed to enable fixed-point code generation later. Guidelines exist to assist in the transformation. Figure 1 illustrates such a guideline which will be used as a running example throughout the rest of the paper.

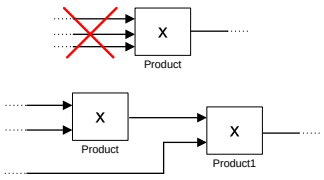

Figure 1: Guideline Violation and Correction

The presented guideline forbids the use of product blocks with more than two operands (cf. upper part of the figure). The multiplication of more than two operands produces

intermediate results. When generating fixed-point code for a model, scaling information for the intermediate results is needed, which cannot be determined automatically. Thus, product blocks with more than two operands have to be transformed into a cascade of product blocks with two operands each and proper scaling information (cf. lower part of the figure).

Violations of this guideline can be detected automatically as will be shown in the following sections. A fully automated correction is not possible, because of the missing scaling information. However, the transformation of the structure, i.e., the creation of the cascade, can be done automatically.

3. TOOL ARCHITECTURE

We developed a prototype for the automatic detection of guideline violations in Matlab/Simulink models based on the FUJABA TOOL SUITE and some extensions (cf. Figure 2). *Matlab/Simulink* is equipped with a *Java Virtual Machine*. A *Matlab/Simulink M-Script Interface* allows Java applications running inside the virtual machine to execute M-Script commands and thus to access the *Matlab/Simulink Model*. FUJABA is started from within *Matlab/Simulink* and runs in its virtual machine.

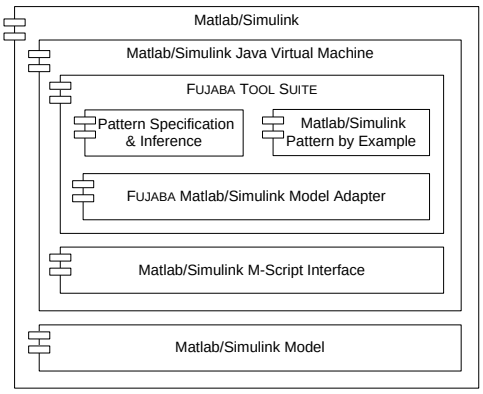

Figure 2: Tool Architecture

Guideline violations are specified formally in FUJABA by so-called pattern rules w.r.t. a Matlab/Simulink metamodel. An inference algorithm applies the rules to the model under analysis. The specification and inference is based on an approach for the recognition of design pattern implementations [6] which has been developed at our group. The *Pattern Specification & Inference* component represents the implementation of this approach by several FUJABA plug-ins which are reused by our prototype and remain unchanged.

Basically, guidelines are specified as pattern rules in abstract syntax based on the Matlab/Simulink metamodel. In addition, the *Matlab/Simulink Pattern by Example* component facilitates the derivation of pattern rules from examples given in Matlab/Simulink notation. The derived pattern rules are able to recognize structures which are equivalent to the example models in arbitrary models.

Both components use an implementation of a rudimentary Matlab/Simulink metamodel provided by the FUJABA *Matlab/Simulink Model Adapter*. The implementation conforms to FUJABA's modeling and implementation conventions and allows direct read access to the model inside the running Matlab/Simulink instance using the *Matlab/Simulink M-Script Interface*. The model adapter component allows to start the pattern inference on the Matlab/Simulink model from within FUJABA and also supports the display and highlighting of detected guideline violations in Matlab/Simulink.

4. GUIDELINE CHECKING

Guideline violations are specified by so-called pattern rules on top of a metamodel. The metamodel is presented in Figure 3 and acts as an adapter to the model in a running Matlab/Simulink instance. A *Model* is used as a root element for the pattern detection and consists of different *BlockDiagrams* which contain different *Blocks*. A *Block* is either a *Constant*, a *Product* or a *Display* block. A *Block* can be connected to another *Block* by a *Line* using the *source* and *target* associations. A selected *BlockDiagram* which is in the foreground of Matlab/Simulink can be accessed using the *current* association.

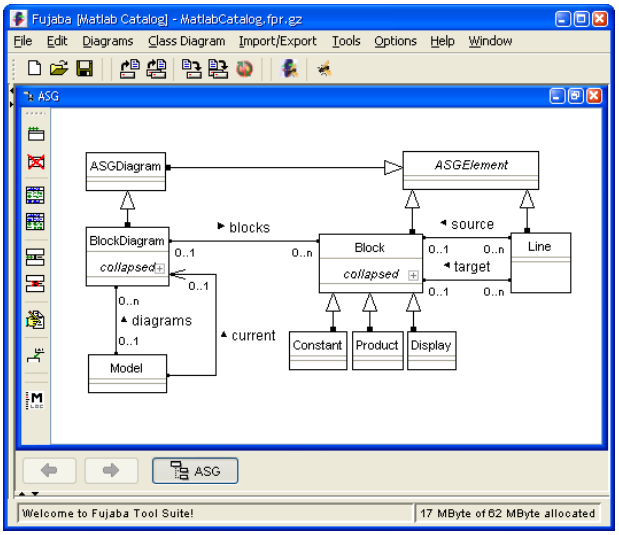

Figure 3: Metamodel

The pattern rules are based on graph grammar rules. Figure 4 gives an example of a pattern rule which detects violations of the example guideline presented in Section 2. The left-hand side of a rule requires a certain structure of metamodel elements to be present in the analyzed model. In this case, there has to be a product block that is connected to three lines which have the block as target. In addition, each of those lines is required to have a block as source.[1] If the left-hand side of a pattern rule can be found, its right-hand side creates an annotation object and links it to selected model elements to mark the pattern occurrence. In the example pattern rule, an annotation named *ProductWithMoreThan2Operands* is created and linked to the product block.

Pattern rules may require annotations created by other rules in their left-hand side thereby enabling composition and reuse of sub patterns. In addition, certain elements (including annotations) may be marked as triggers. In the example rule, the product block has been marked as a trigger and thus is displayed with a bold border. For each element in

[1]Note that the specification of the three source blocks is not really necessary here. However, they might be used to impose additional constraints.

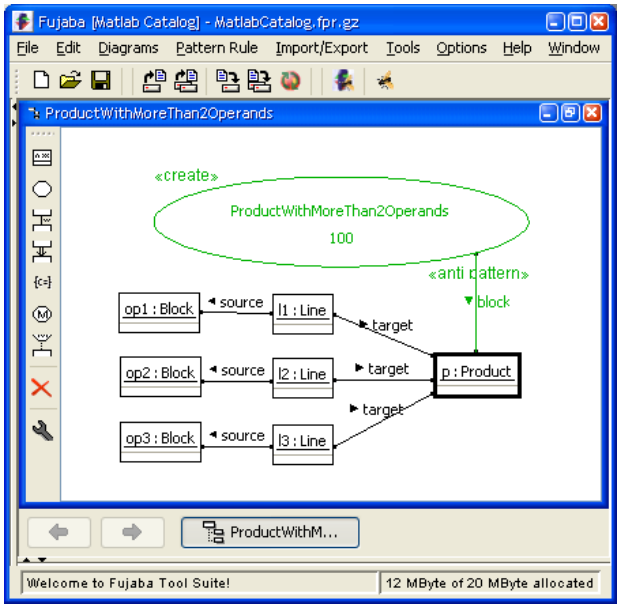

Figure 4: Guideline Violation Specification

the model that corresponds to a trigger – in this case each product block – the pattern rule is applied. By specifying annotations as triggers, the successful application of pattern rules may trigger the application of higher level rules. The actual application of pattern rules is controlled by an inference algorithm. The algorithm uses dependencies between the pattern rules and the triggers to determine which rule is applied when. This approach facilitates the analysis of even huge models which is essential since Matlab/Simulink models are expected to contain between 10.000 and 100.000 elements.

In order to exploit FUJABAs support for pattern detection and annotation, it is necessary that the metamodel is implemented according to the rules of FUJABA. The compliance with these rules allows navigating between model elements, accessing and modifying model elements, as well as creating new model elements. These kind of requirements are fundamental for FUJABA's graph rewriting algorithm - if a metamodel implementation does not fulfill these requirements, FUJABA's graph rewriting algorithm will not work with that model at all. Furthermore, the pattern detection requires the metamodel classes *Block* and *Line* to inherit from FUJABAs built-in class *ASGElement* and *BlockDiagram* to inherit from the built-in class *ASGDiagram*, respectively.

In Matlab/Simulink, the metamodel does not fulfill the requirements needed by FUJABA and it can obviously not be changed. In order to realize the checking of pattern rules anyway, we have implemented a rudimentary FUJABA compliant metamodel for Matlab/Simulink. From this metamodel specification, we generate the metamodel implementation using FUJABA's code generation facilities. The automatic code generation guarantees a FUJABA compliant metamodel that fulfills the requirements for FUJABA's graph rewriting algorithms by providing appropriate methods for the access of attributes and associations.

Up to this point, the automatically derived metamodel implementation does not provide any access to the models of Matlab/Simulink. However, this is a key requirement for the detection of guideline pattern rules in a running Matlab/Simulink instance. To establish such a linkage to the Matlab/Simulink models, we have to extend the generated metamodel to a model adapter with direct access to Matlab/Simulink models using the adapter pattern [3].

For this purpose, we remove all attributes from the generated metamodel classes and replace the generated implementation of the access methods manually using the Matlab/Simulink M-Script Interface. This built-in Application Programming Interface (API) provides direct access to Matlab/Simulink models in a running Matlab/Simulink instance. Note that we replace the method implementations without changing the method signatures in order to preserve the interface to FUJABA's graph rewriting facility. Instead of manually changing the generated implementation, another option would be to adapt the code generation. However, the implementation of the metamodel will not be generated very often and it is not clear yet how homogeneous the access method implementations will be and if an adaption of the generator is feasible.

The resulting model adapter implementation is a stateless model adapter with lazy object initialization, i.e., the adapter objects are created only on demand. Additionally, the realized adapter keeps a list of already adapted model elements and reuses them each time the model element is revisited. Thus, a model element is always represented by the same adapter object and adapter object identities are preserved. Furthermore, this guarantees a fast access to already adapted model elements of the Matlab/Simulink model and reduces at the same time the number of needed adapter objects. Of course, in the worst case, i.e., if all model elements have to be examined, this will result in one adapter object for each adapted model element anyway.

The current prototypical implementation of the model adapters handles only the metamodel shown in Figure 3. That is, of course, only a small fraction of the original metamodel. As well, the current Matlab/Simulink Model Adapter provides only read access to Matlab/Simulink models and thus, modifications to a Matlab/Simulink model are not possible yet. However, these extensions are intended as future work.

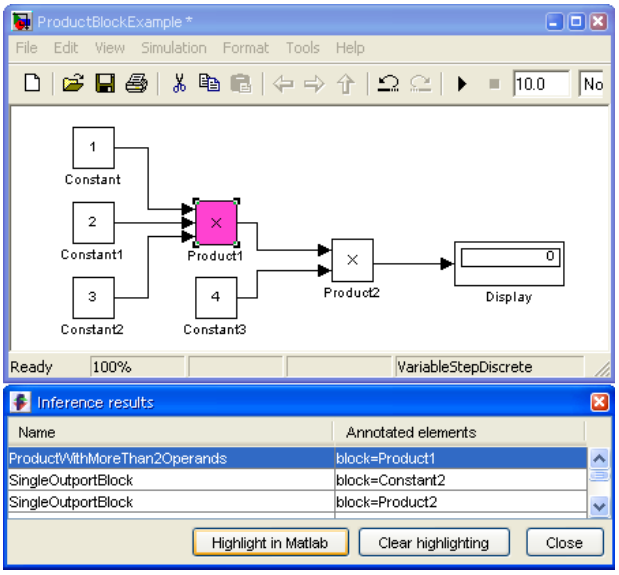

Figure 5: Result Viewer and highlighted Violation

The pattern inference obtains a *Model* instance from the Matlab/Simulink Model Adapter. Starting from the *Model* instance, model elements that trigger pattern rules are collected and the corresponding rules are applied. Each successful application of a pattern rule creates an annotation marking the pattern occurrence, i.e. a guideline violation, in the Matlab/Simulink model. The lower part of Figure 5 shows a result viewer provided by FUJABA that displays all created annotations. In addition, the user may select an annotation and choose to highlight the annotated model elements. In Figure 5 the *ProductWithMoreThan2Operands* annotation has been selected and the annotated product block with more than two operands has been highlighted and selected in the Matlab/Simulink editor.

5. SPECIFICATION BY EXAMPLE

In most cases, guidelines are not static - new guidelines have to be added or existing ones have to be adapted to special user, enterprise, project, or domain specific requirements. In the previous section, guidelines are specified as pattern rules using the abstract syntax defined by the metamodel of Matlab/Simulink. Although the presented approach enables the formalization of guidelines and the detection of their violations, a disadvantage of the approach is that the developer of a guideline rule has to be familiar with the used pattern language and the metamodel of Matlab/Simulink. However, since the complete metamodel of Matlab/Simulink is quite large and complex, this can also lead to large and complex pattern rules that are hard to read and maintain. As a consequence, it becomes infeasible for the user to change the set of guideline rules easily.

To overcome this problem, we developed an approach that enables a user to model an example for a guideline violation in the concrete syntax of Matlab/Simulink. The given guideline violation example is automatically transformed to a pattern rule in abstract syntax notation. The automatically derived pattern rule can be used for the detection of guideline violations in arbitrary models using FUJABA's inference algorithm as shown in the previous section.

Figure 6 shows an example for a guideline violation and the pattern rule as a result of the transformation. In the lower left corner of Figure 6 the user specified a guideline rule by giving an example for its violation in the concrete syntax of the Matlab/Simulink model. The presented guideline discourages from using a product block for the multiplication of two constants since this will result once again in a constant. This sort of calculations is unnecessary since the result of the multiplication can be calculated in advance. Hence, the product block and its operands can be replaced by one constant block holding the calculated result. This leads to smaller models and increases the performance of the overall calculation.

The presented example guideline violation is transformed automatically to the pattern rule shown in the background of Figure 6. The pattern consists of two objects representing the constant blocks and one object for the product block. The connections between the constant blocks and the product block are represented by line objects and appropriate links between them. Additionally, an annotation object is created and connected by links to objects representing blocks. The name of the annotation object is derived from the name of the example diagram. Note that in the guideline violation example, the product block is selected. Therefore,

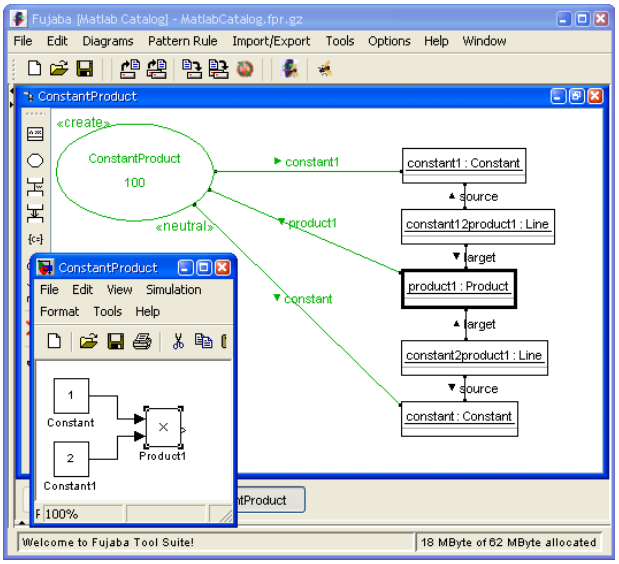

Figure 6: Guideline Specification by Example

the object that represents the product block in the pattern rule is marked as a trigger for the inference algorithm, i.e., it has a bold border in the pattern rule specification.

The transformation is realized using the technique of triple graph grammars [8]. The realized model transformation for the prototype handles the simple metamodel from Figure 3 and comprises five rules which are executed utilizing two FUJABA plug-ins [9]. For our example guideline specifications, these transformation rules generate valid pattern rules automatically. However, since our approach is still a prototype, we recommend the rule developer to inspect the extracted pattern rules and – if necessary – to adapt them manually in order to get a valid guideline violation pattern. In the future, we will study more guidelines and try to specify them by examples in order to get more experience with this approach.

6. CORRECTING VIOLATIONS

Guideline violations usually have to be corrected. How a violation has to be corrected and if this correction may be done automatically, depends, of course, on the particular guideline. In case of the example guideline presented in Section 2, a single product block with more than two operands has to be transformed into a cascade of product blocks having only two operands each and with appropriate scaling information. A fully automated correction is not possible. However, the transformation of the structure into a cascade can be automated.

We propose to formally specify corrections by graph transformations. Figure 7 shows such a transformation for the example guideline specified by a story diagram [2]. The story diagram expects a product block as argument which has been detected to violate the guideline. The given block is bound to the *product* object in the single story pattern. Next, one outgoing and three incoming lines of the *product* block are bound (objects *lOut*, *l1*, *l2*, and *l3*, respectively). If the binding is successful, a new product block (*newProduct*) is created. One of the incoming lines (*l3*) as well as the outgoing line *lOut* are reconnected to the new product block and a new line *lNew* connecting the original with the

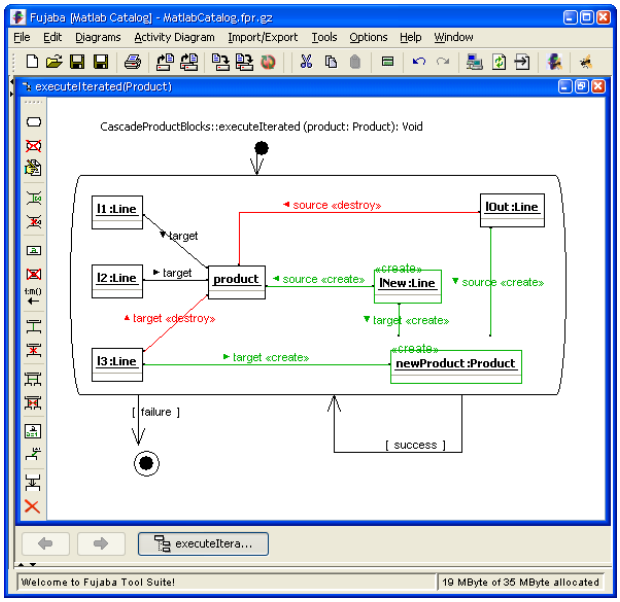

Figure 7: Partial Correction of a Violation of the Example Guideline

new product block is created.

Thus the result of the original product block and one of its former operands become the two operands of the new product block, its result is passed to the former receiver and the original product block has one fewer operand. The story pattern is repeated while the original product block has still at least three operands. Each application decreases the number of operands by one and thus the product block has only two operands left upon termination.

A semi-automatic correction of violations of the example guideline is possible. For a violation of the constant product guideline mentioned in Section 5, a correction would be to replace the two constants and the product block by one constant block with the product of the two original constants as value. This can be automated as well. However, there are many more guidelines and an analysis will be necessary in order to identify more transformations for correcting violations. It has to be analyzed if those transformations can be specified by graph transformations/story diagrams as well. In addition, the question whether the application of a transformation to a model results in a correct model again, is not addressed here but investigated in a different context (cf. [5]).

Furthermore, transformation specifications are not executable in our prototype yet because of two reasons. First, our metamodel implementation does not support write access to a Matlab/Simulink model yet. Secondly, a user interface for applying transformations to detected violations is missing.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a prototype for detecting guideline violations in Matlab/Simulink. The novelty of the presented approach is that it is based on formal pattern rules which enable the formalization of so far informally described guidelines. In order to ease these specifications our approach allows specifying guideline violations as examples in the concrete syntax of the Matlab/Simulink modeling language. Furthermore, we have proposed to use graph rewriting for the automatic correction of guideline violations.

Up to now, the prototypically realized Matlab/Simulink adapter implements only a very small fraction of the Matlab/Simulink metamodel. In addition, the adapter does not support modifications of the model which is required for the automatic correction of guideline violations. Hence, our plan for the future is to extend the metamodel and the capabilities of the provided adapter and to realize the automatic correction based on graph rewriting.

The presented specification of guidelines using examples in the concrete syntax of the modeling language is quite promising. However, an extracted pattern rule reflects exactly the specified example. In practice, this can lead to a large number of quite similar guideline rules. To solve this problem, we have to study how a more abstract and general pattern rule can be extracted from a set of examples.

8. REFERENCES

[1] U. Eisemann. Guidelines for a model-based development process with automatic code generation. In *Preliminary Proceedings of the Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 3), Heinz Nixdorf MuseumsForum, Paderborn, Germany. October 13 and 14, 2005*, number tr-ri-05-261 in Technical Report, Department of Computer Science, University of Paderborn, Paderborn, Germany, October 2005.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer-Verlag, 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] The MathWorks. *MATLAB Model Advisor*, 2006. http://www.mathworks.com/products.

[5] M. Meyer. Pattern-based Reengineering of Software Systems. In *Doctoral Symposium of the 13th Working Conference on Reverse Engineering, Benevento, Italy*, October 2006. (to appear).

[6] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[7] Ricardo, Inc. *MINT*, 2006. http:/www.ricardo.com/mint.

[8] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20*th *International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.

[9] R. Wagner. Developing Model Transformations with Fujaba. In H. Giese and B. Westfechtel, editors, *Proc. of the* 4^{th} *International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*. University of Paderborn, September 2006.