Checking and Enforcement of Modeling Guidelines with Graph Transformations

Carsten Amelunxen[1], Elodie Legros[1], Andy Schürr[1], and Ingo Stürmer[2]

[1] Darmstadt University of Technology, Real-Time Systems Lab,
[amelunxen|legros|schuerr]@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de
[2] Model Engineering Solutions, Berlin,
stuermer@model-engineers.com
http://www.model-engineers.com

Abstract. In the automotive industry, the model driven development of software for embedded controller units evolves to become the standard paradigm. In this domain, the development is based on executable block diagrams and StateCharts which are provided by the commonly used tool MATLAB Simulink/Stateflow. 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. Checking these guidelines and eliminating detected violations manually during audits is a tremendous amount of boring work. In this paper, we show how graph transformations can be used to automate the process of guideline checking and the execution of repair actions. Based on our experiences in an industrial context, we discuss the pros and cons of graph transformations compared to other specification approaches and we finally present a proposal how to combine graph transformations with other modeling paradigms as the most promising approach.

1 Introduction

Nowadays, model-driven development is common practice within a wide range of automotive embedded software development projects. In this domain, the standard modeling language UML still plays a neglectible role and the *MathWorks MATLAB Simulink/Stateflow (MATLAB SL/SF)* [MAT] environment is used as a de facto standard. Simulink supports a block-oriented style of modeling that combines the data-flow programming paradigm with differential equation solvers; Stateflow adds a discrete event and state-oriented style of modeling based on Harel's concepts of hierarchical automata (StateCharts).

Embedded controller software is either manually developed by programmers using Matlab SL/SF models as executable requirements specifications or generated automatically by code generators which translate Matlab SL/SF models into rather efficient C code. In both cases the reliability, robustness, and efficiency of the developed code heavily depends on the quality of the specified models.

Therefore, generally accepted modeling guidelines – such as the MathWorks Automotive Advisory Board (MAAB) guidelines – are usually adopted. These modeling guidelines are either manually or automatically checked during audits using tools like the Mathworks Model Advisor. However, for huge models, this can add up to a few hundreds or even thousands of violations that must be corrected manually by the modeler.

A recent in-house study at DaimlerChrysler showed us that automated and partly interactive model corrections can reduce the effort of model refactoring activities up to 70 percent. Nevertheless, we are not aware of any tool support in this direction except of our *Matlab SL/SF Model Analysis and Transformation Environment MATE*. MATE has been developed in a joint effort of four universities and two companies [SDG+07]. The main motivation for starting the MATE project was our observation that the implementation of modeling guidelines today takes place on a very low level of abstraction using imperative programming languages. Therefore, the realization of really complex checks is almost infeasible as well as the development of even more complex model transformations that eliminate identified guideline violations automatically. It is our impression and the intention of this paper to show that, in general, graph transformations offer significantly better support for the specification and implementation of modeling guidelines and refactorings. Furthermore, we will discuss the pros and cons of graph transformations compared to a limited number of other specification paradigms. We will conclude the paper with a proposal how to extend graph transformations to overcome some limits that still impact their usefulness in this application domain.

The rest of this paper is, therefore, organized as follows: section 2 discusses the motivations for this work and the MATE project in general, whereas section 3 compares the MATE environment with other MATLAB SL/SF guideline checking frameworks and points out the highlights of a graph transformation based approach. Section 4 then introduces a representative set of guidelines as running example and explains the overall structure of MATLAB Simulink models. Afterwards, section 5 discusses the specification of some guidelines using a mixture of regular expressions and 1st order logic expressions, whereas section 6 then presents some graph transformations that specify the selected guidelines and appropriate repair actions where possible. Finally, section 7 summarizes the results of the comparison of different specification paradigms and discusses our plans for future work concerning the design and implementation of a more powerful graph transformation environment.

2 The MATE Project

The MATE project [SDG+07] provides support for semi-automatic checking and enforcement of modeling guidelines as well as for version management, design pattern instantiation, and interactive model refactoring and beautifying operations. It is a joint project of two companies (DaimlerChrysler, Model Engineering Solution) and four universities (Technical University of Darmstadt, University

of Kassel, University of Paderborn, University of Siegen). This project was born out of an urgent need of the automotive industry for more sophisticated tool support in this area. Automotive software developers using MATLAB SL/SF are confronted with the same well-known and ordinary maintenance and quality assurance problems of everyday life programming. Due to the fact, that the main application domain of MATLAB SL/SF models is the simulation and code generation for safety-critical embedded systems, the importance of quality assurance becomes even more significant. A MATLAB SL/SF code generator may only produce high quality C code if its input models are of high quality, too.

Therefore, rigorous model audits (review processes) play an important role for a model-driven automative software development process. The significance of model reviewing is supported by a case study [SCFD06] which presents 146 critical model changes due to findings from a multi-iterative reviewing process on a model of 9308 blocks. All in all, the reviewing took a netto time of 1600 minutes, nearly 27 hours. Since, reviewing is a time-consuming and thus cost-intensive process, it is a highly desirable task to ensure the quality of a model already during its creation and, thereby, reduce the efforts of reviewing. This can be done by formulating a set of modeling guidelines that are checked continuously and automatically earlier on during the model development process.

Therefore, modeling guidelines for MATLAB SL/SF are very popular in the automotive industry. There are e.g. modeling guidelines provided by the MathWorks Automotive Advisory Board (MAAB) [MAA]. These guidelines focus on several aspects like naming and graphical layout conventions, tool and model configurations, logical errors, forbidden design anti-patterns and recommended design pattern, and so on. In fact, most of these guidelines imply one or more repair actions, which often can be executed automatically or semi-automatically with some degree of user feedback. We expect, based on practical experiences, that from all captured guideline violations approximately

- 45% can be eliminated automatically
- 43% can be fixed with additional user input
- 4% can only be removed manually
- 8% are not classified yet

Analysis as well as refactoring of MATLAB SL/SF models demands full access to MATLAB's model repository. Such an access is provided by an API written in M-Script, a proprietary script language. Both the used C-like scripting language and the tool's API evolved over many years. As a consequence, it takes quite some time and efforts to learn how to program reliable model checks and transformations using this approach. The MATE project overcomes these problems by providing a layer of uniform API adapters on top of which visual graph queries and transformations can be developed on a considerably higher level of abstraction. First experiences indicate that encoding new guidelines on this new level of abstraction reduces the needed efforts up to a factor of four and results in definitely more readable code as you will see later on.

The MATE project right now uses the Fujaba graph transformation tool [Fuj] and its meta-modeling plug-in MOFLON [MOF] to specify the needed

graph queries and transformations. Generated Java code either directly manipulates MATLAB SL/SF models via the tool's API or works on an offline model repository. Both solutions have their specific pros and cons: working directly on the tool's API is the preferred solution, when interactive model refactoring and beautifying operations have to be implemented. Working with an offline repository with special indexes has some advantages, when complex analysis operations have to be executed. A more detailed description of the MATE system's architecture and its functionality as well as its integration with the MathWorks tool suite is out-of-scope of this paper.

3 Related Work

Well-known examples of other MATLAB SL/SF analysis tools are MathWork's own Model Advisor [MAT] and MINT [Min]. Both tools rely on the execution of MATLAB M-Scripts to identify modeling rule violations within Simulink and Stateflow models. As already mentioned, this approach requires intimate knowledge of a tool API that has been developed over a period of many years always having backward compatibility in mind. Furthermore, a concise description of the abstract syntax and (static) semantics of manipulated modeling language instances does not exist and has to be inferred step by step by testing the functionality of one API operation after the other. Furthermore, M-Script developers have to use imperative programming constructs for data flow analysis, pattern matching and rewriting activities – a very error-prone and time-consuming task.

Therefore, the MESA project [FR07] started to develop an own meta model for the modeling languages Simulink and Stateflow that captures the abstract syntax and part of the static semantics of these two languages. This meta model is used to generate an offline model repository for guideline checking purposes (as we do in the MATE project). Guidelines are specified in the logic-based Object Constraint Language OCL of the OMG. The specification of refactoring operations is still out-of-scope due to the fact that OCL does not offer any support for modifying models. Similarly, the GME/GReAT team developed a MATLAB SL/SF meta model with the intention to specify model/graph transformations that either create or translate MATLAB SL/SF models [NKS+05]. As far as we know, guideline checking and repair actions have not yet been addressed. Furthermore, the experiment reported in [MSD06] describes a successful use of graph transformations for the detection and resolution of UML model inconsistencies.

Despite of the obvious deficiency of OCL compared to graph transformation languages, it is not clear whether a logic-based textual language like OCL or a visual rule-based approach as offered by Fujaba/MOFLON or GReAT is more appropriate for the specification of modeling guidelines. It is the main purpose of this paper to start a systematic comparison of both specification paradigms. We will see later on that both approaches have their pros and cons and should be combined with other concepts to obtain a more powerful (meta) modeling and specification language.

4 Modeling Guidelines for MATLAB Simulink

This section introduces our running example, four different guidelines for MATLAB Simulink models. These examples constitute a representative set and are well-suited for the comparison of different specification paradigms. Furthermore, we present a considerably simplified meta model of MATLAB Simulink.

4.1 Guidelines

In the following, we discuss guidelines that involve string pattern matching, calculation of complex arithmetic expressions as well as local and global pattern matching operations. Aspects concerning a proper model layout are out of scope of this paper, but can be handled in similar ways.

Guideline 1: Naming of Subsystems. Usually modeling guideline catalogues list quite a number of naming conventions that impose certain restrictions on the name/identifier of a single modeling element. Often these restrictions are intended to increase the readability of a model or to forbid the usage of identifiers that are known to cause troubles during code generation. We have selected a typical naming convention for Simulink subsystems. The name of a subsystem may consist of lower and upper case alphabetic characters, including numeric characters and underscore. There must neither be more than one consecutive underscore nor an underscore at the beginning or the end of a subsystem's name. Number are also forbidden as first character of a subsystem's name.

Guideline 2: Naming of Enable Port block. The second modeling guideline concerns the naming of `Enable Port` blocks. An enable block of a subsystem permits or blocks its execution depending on the signal that is processed by this block. In order to be able to identify enable port blocks immediately, a guideline demands that the `Enable` block's name matches the name of the corresponding enable signal of the regarded subsystem (cf. Fig. 1).

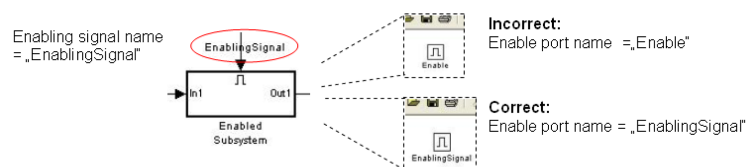

Fig. 1. Naming restrictions for Enable Port block

Guideline 3: Unconnected signals. This guideline ensures that every element of a Simulink model is connected. Unconnected subsystems, basic block inputs, outputs or unconnected signal lines are not allowed. Thus, this rule is rather important for the structural correctness of a model. A violation of this rule inevitably leads to an erroneous model (cf. Fig. 2). Nevertheless, a violation of this guideline is quite easy to fix by connecting the unconnected inputs to ground blocks and the unconnected outputs to terminator blocks (if affected inputs and outputs are not needed during model execution).

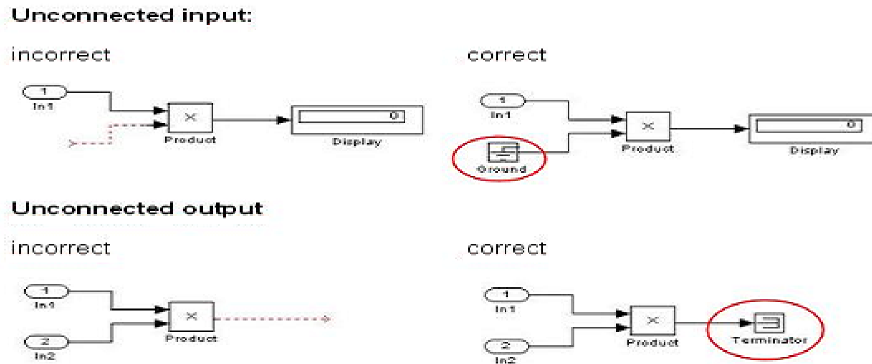

Fig. 2. Restrictions concerning unconnected signals

Guideline 4: Numerical and Dataflow Analysis. It is possible to design control and signal processing systems that will be implemented using fixed-point arithmetic. Two drawbacks must be considered, when using fixed-point arithmetic [Soh06]. The first one is the risk to introduce new sources of overflow/underflow (compared to a floating point arithmetic execution of the same model). An overflow/underflow occurs, when a calculation produces a result greater/smaller than the number that can be stored in the specified fixed-point format. The fixed-point representation is defined by the number of bits used and the scale factor which corresponds to the least significant bit in base 2. A number x in fixed-point arithmetic with n bits and a scale factor S_x must be a value in the range of

$$-2^{n-1} \cdot S_x \leq x \leq (2^{n-1} - 1) \cdot S_x \tag{1}$$

if signed or in the range of

$$0 \leq x \leq (2^n - 1) \cdot S_x \tag{2}$$

if unsigned. In case of overflow or underflow, the signal is distorted, the error is propagated to the output without the user knowing it.

The other drawback are new rounding errors due to the loss of precision and the quantification error caused by fixed-point arithmetics. A small output error usually can be ignored, but further processing and propagation of numerical errors often leads to problems. That is why we need analysis rules that compute lower and upper bounds for computed fixed-point arithmetic values as well as conservative estimates for rounding errors using interval arithmetic. In some cases precision problems can be easily fixed automatically by increasing the scale factors of affected blocks. In other cases, complete subsystems have to be restructured manually. Furthermore, it is possible to automatically rewrite Simulink models such that they are able to handle overflows and underflows either by simply truncating output values using `SaturateBlocks` or creating signals that trigger later on manually added error-handling computations.

4.2 MATLAB Simulink Metamodel

In the past quite a number of meta models for MATLAB Simulink have already been developed. Most of them are quite simple and introduce a rather generic abstract syntax model with a small number of concepts. 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`. Furthermore, blocks have attributes in the form of `PropertyName` pairs that are either atomic or consist of properties in turn.

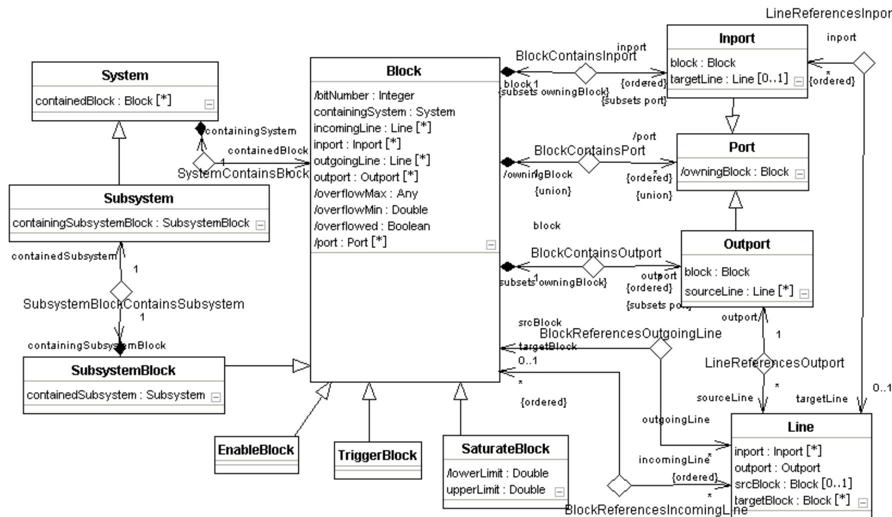

Fig. 3. Simplified metamodel of Matlab Simulink

Such a meta model does not contain any information about required or optional properties of certain types of blocks. It simplifies the development of import and export functions for a model repository or the implementation of an API considerably, but it is not very useful for the specification of guideline checks and repair actions for the following reasons: accessing specific property values of blocks of specific types is very awkward and error-prone. To solve this problem a meta model must be introduced, where each block type is defined as a separate meta class with its properties listed as (meta) attributes of this meta class. Furthermore, we often have to navigate from one `Block` to another one identifying first the right `Inport` or `Outport`, following then a list of `Lines` to the opposite `Outport` or `Inport`, and finally traversing the link from this element to its own `Block`. Last but not least, when writing analysis rules, we often have to identify patterns, where a block has a certain combination of property values or a certain number of outgoing or incoming connections.

For these purposes we have added another meta model layer on top of the generic Simulink meta model, where we introduce specific block types like `SaturateBlock` with specific properties like `lower/upperLimit` (cf. Figure 3). These model elements represent *derived data* that is automatically computed

using the generic meta model elements as input. In fact they are a kind of *updatable view* that can be used to specify rather compact and readable model queries and transformations. A transformation that creates e.g. a single virtually existing SaturateBlock object with appropriate property values in reality creates a generic `Block` object with two associated `Property` objects for its two attributes `lower/upperLimit`.

Furthermore, we are using derived attributes and associations that are needed for data flow analysis purposes or simplify navigation between connected blocks. As far as we know this is the first time that a MATLAB SL/SF processing tool combines in this way a simple generic meta model with a rich meta model that lists all needed types of blocks with their specific properties. Right now, Java code is used to implement the derivation rules that keep both meta modeling layers in a consistent state. We are planning to use a language like OCL for that purpose for reasons that will be explained later on and to develop in general more sophisticated support for the definition of updatable views on models (graphs) as discussed in [JKS06].

5 Guideline Specification without Graph Transformations

It is state of the art that MATLAB SL/SF modeling guidelines are implemented using the imperative scripting language M-Scripts. Furthermore, in the model-driven software engineering community an increasing number of projects is starting to use OMG's logic-based language OCL for the definition of integrity constraints and static semantics rules for models and meta models. Therefore, we will present in this section one example of an M-Script implementation as well as a number of examples of OCL specifications of the selected modeling guidelines. In the following section we will then present graph transformation specifications of the same set of guidelines for comparison purposes.

5.1 M-Script

Today almost all modeling guideline checks are implemented using the programming language M-Script that is part of the MATLAB SL/SF tool suite – on a very low level of abstraction. In the following we present the M-Script implementation of the analysis of guideline 2 as an example. We skip the detailed explanation since the example is serving its purpose of giving an impression of what M-Script checks are suffering from.

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.

```
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
```

5.2 Regular expressions

Since consistent naming is a very important feature of high quality MATLAB SL/SF models, an approach replacing M-Script as first choice approach has to provide regular expressions for the description of string restrictions. Regular expressions provide a technique to describe legal sets/languages of strings based on syntactical rules only. Thus, regular expressions cannot act as a substitution of M-Script. They rather provide a powerful addition to an existing guideline implementation approach. In the following, we demonstrate the usefulness of regular expressions by the implementation of guideline 1.

The pattern which is intended by guideline 1 is formulated in the syntax of regular expressions. Then, the negation of this pattern is used to detect guideline violations. Since the name of a subsystem can neither start with an underscore nor a number, the name must start with an alphabetic character, which is represented by the term [**A-Z a-z**]. Furthermore, the rest of a subsystems's name consists of an arbitrary number of alphabetical characters and numbers which must not be separated by more than one consecutive underscore and must not end with an underscore. Thus, the following regular expression matches a correct subsystem name:

```
[A-Z a-z](([A-Z a-z 0-9]*)(\_?)([A-Z a-z 0-9]+))*
```

5.3 The Object Constraint Language

The application of the Object Constraint Language (OCL) provides an approach which could in general act as a basis for the formalization of all kinds modeling guidelines. OCL is a precise logic-based language which provides constraint and object query expressions on MOF/UML compliant models or meta models. Since modeling guidelines represent constraints on model elements or relations between

model elements which have to be respected, OCL can be used for a formal description of such rules. In the following, we demonstrate the application of OCL by the implementation of guideline 2 and 3. In case of guideline 2, the two different cases of unconnected lines and unconnected ports have to be considered. Both can be covered by OCL invariants in different contexts. First of all, the following invariant applies in the context of a line, stating that a line must have one source and one target block.

```
context Line
inv: (srcBlock != null) and (targetBlock != null)
```

Furthermore, a port has to be connected to a line. Since the classes `Inport` and `Outport` are connected to the class `Line` by different associations, we have to write two different constraints for the two regarded classes. Both invariants are listed in the following.

```
context Inport                      context Outport
inv: targetLine != null             inv: sourceLine != null
```

As a consequence guideline 3 is formalized by a set of three OCL invariants. In fact, all three invariants are quite trivial and a tremendous improvement compared to the corresponding M-Script implementation presented above. The presented OCL specification has only one drawback: a single modeling guideline is translated into three different constraints instead of being a single piece of code. If a one-to-one correspondence of guidelines and constraints is an issue (e.g. for reasons of maintainability of guideline implementations) then we can resort to the following solution, where a single more complex OCL constraint enforces the same guideline.

```
context Block
inv: incomingLine->forAll( srcBlock != null ) and
     outgoingLine->forAll ( targetBlock != null ) and
     inport->forAll(targetLine != null) and
     outport -> forAll(sourceLine != null)
```

The OCL expressions presented above probably give the reader the impression that it is straight-forward to produce and to understand logic-based specifications of modeling guidelines. But this is no longer true, when more complex patterns have to be specified. Let us consider our modeling guideline 2. This guideline requires that the enable block name matches the name of the signal enabling the subsystem. The class `SubsystemBlock` that contains both the regarded block and its corresponding signal is an obvious choice as context for the to be defined OCL expression.

First of all, we have to check that the regarded subsystem contains an `EnableBlock`. Then two elements of the subsystem must be determined and compared: the name of the enabling signal and the name of the corresponding enable block. To compute the name of the enabling signal, we must match that instance of the class `Line`, whose value of `PropertyName` “DstPort” is equal to “enable” and return its name (cf. subexpression starting at label (1) below). To find the name of the enable block, we must select the block instance of the class

EnableBlock contained in the subsystem and return its name (cf. subexpression starting at label (2) below).

Please note that a subsystem neither may contain more than one enable block or more than one enabling signal. That means that the intersection of the computed sets of signal and block names is either the single common name (the guideline is respected) or empty (a violation of the guideline).

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

This example clearly shows that 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, it is almost unfeasible to encode guideline 1 or guideline 4 using OCL. In the first case the pattern matching facilities of regular expressions are missing, in the second case we are running into problems, when we have to compute intervals of possible value ranges as well as upper bounds for rounding errors. OCL offers some basic operators on integers and reals for that purpose, but does not directly support more complex arithmetic operations like the calculation of two to the power of a negative value. It is, therefore, necessary to delegate these computations to a host programming language via method calls embedded in OCL expressions. As a consequence, we will not present a specification of guideline 4 here.

6 Analysis and Refactoring with Graph Transformations

In this section we finally present graph transformation specifications of our guidelines. For this purpose the visual SDM (story driven modeling) diagram syntax is used [Fuj] that is supported by the graph transformation tool Fujaba and our plug-in MOFLON [MOF]. Each of these specifications relies on the existence of a context/parameter object (as the OCL expressions presented before) and it is evaluated for all objects of the regarded context class.

The first specification presented in Fig. 4 consists of four different SDM activity diagram nodes: a start node follwed by a pattern node containing a single object with a complex attribute condition, followed by an action node with a piece of Java code and a terminate node. The presented graph pattern checks the `qualifiedName` attribute of a given `SubsystemBlock` using a regular expression. The check either succeeds and the execution of the small SDM activity diagram terminates or it fails. In the latter case an external Java method

is called that logs the detected modeling guideline violation. Please note that SDM diagrams right now do not directly support regular expression checking as suggested in Fig. 4. The actual specification uses a work-around based on regular expression-handling mechanisms that are available in the Java programming language. Fortunately, SDM diagram nodes may contain an arbitrary piece of Java code such as the logging method `logError` used in the regarded diagram.

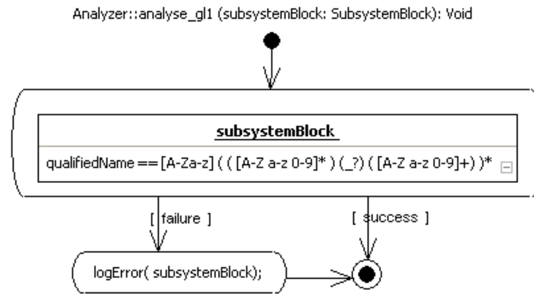

Fig. 4. Graph query for the analysis of guideline 1

Fig. 5 simultaneously checks and fixes violations of guideline 2. 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 grey/green line inside the `enable` object rectangle assigns the name of the matched `enableSignal` to the regarded `enable` object.

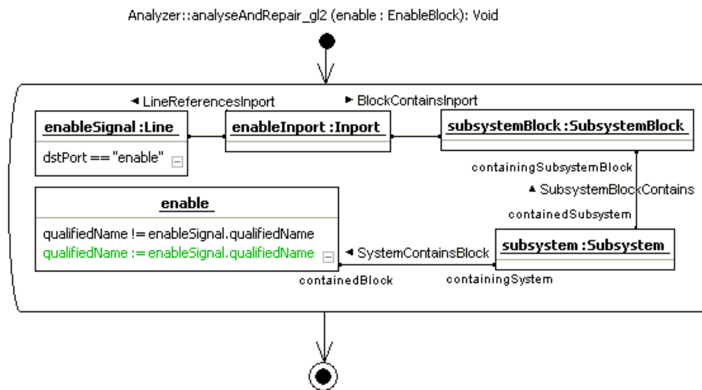

Fig. 5. Transformation that checks and fixes violations of guideline 2

Please compare this specification of the guideline check with an incorporated repair action and the M-Script implementation and the OCL specification presented beforehand. It clearly shows the advantage of graph transformations, when more complex object/link patterns have to be found and modified – at least when we use derived elements to hide certain details of the "real" object structure. In our example, both `qualifiedName` and `dstPort` are derived but nevertheless updatable attributes that are internally represented as separate `PropertyName` objects with a `name` and a `value` attribute. The pretty straightforward code needed for the construction of these updatable views is still hand-

coded in Java. We are working on a new specification approach for updatable views that relies on a special variant of triple graph grammars [JKS06].

The specification of the third guideline is also rather straighforward. It first checks whether the given `inport` object does *not* have an associated `Line` object. If this check fails (and a `Line` object does exist) it then goes ahead and checks for the non-existence of an `Outport` object of the `line` associated with the regarded `inport`. Finally, the programmed graph transformation rule creates the missing objects and links (depicted as grey/green objects and links with stereotype `create`).

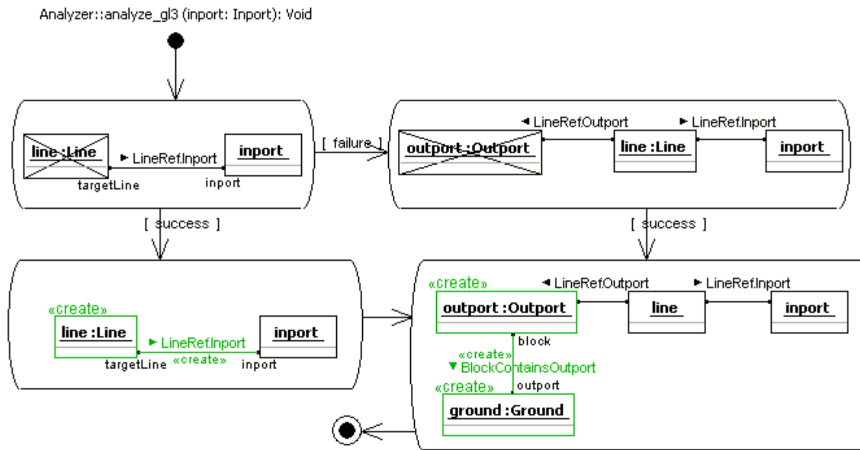

Fig. 6. Transformation that checks and fixes violations of guideline 3

The specification of this guideline is more complex than its OCL counterpart presented beforehand for the following reasons: if we want to create missing objects, which is outside the scope of the OCL expression, then we have to distinguish whether just a `Ground` object or a `Ground` object together with an associated `Line` object is missing.

Finally, we have to translate guideline 4 into an SDM diagram specification, which is the most complex of the selected guidelines. Standard data flow analysis is used to compute lower and upper bounds for block outputs as well as upper bounds for numerical errors. For this purpose derived attributes are used, whose evaluation functions are implemented in Java and not in OCL as originally planned for reasons discussed beforehand. The Java code is a straightforward translation of the directed equations presented in [Soh06] and will be omitted due to lack of space here. The graph transformation program presented in Fig. 7 accesses the derived attribute `overflowed` that signals a potential overflow of the output of the regarded block. It then eliminates the detected numerical problem as follows: first of all the bit size of the directly affected block output is increased and then a so-called `SaturationBlock` is introduced that simply restricts the upper and lower boundaries of the computed value range such that we don't have to modify those blocks that process the regarded output as input. For that purpose the direct connection between `aOutport` and `aInport` is deleted

(objects and links with label `destroy`) and replaced by a new `SaturateBlock` object together with one `Inport` and one `Outport`.

This is – of course – just one possible solution how fix the reported problem. In other cases it is sufficient and feasible to increase the block sizes of all following blocks appropriately. Furthermore, sometimes overflows and underflows have to be detected and reported to error handling submodels. And in some cases, large parts of the affected MATLAB Simulink model must be rewritten. The MATE environment, therefore, offers its users a number of alternatives how to fix an underflow/overflow problem that must be selected interactively. Problems with the precision of computed fixed-point values may be solved in a similar way.

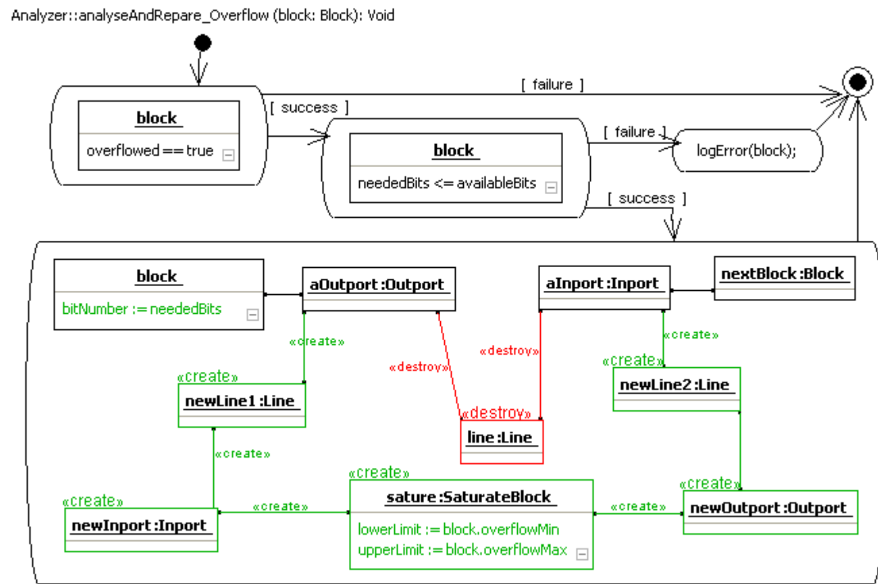

Fig. 7. Graph query for the analysis and reparation of overflows

7 Conclusion

In this paper we have presented specifications of model guideline checking and refactoring operations using a variety of different approaches. In the related discussions we pointed out that neither the logic-based language OCL nor the graph transformation rules of Fujaba/MOFLON are well equipped for handling *all sorts* of modeling guidelines. In principle, SDM diagrams together with their option to insert arbitrary pieces of Java code where needed are an excellent choice for the specification of model analysis and refactoring operations as discussed in this paper.

But, our daily work with the specification of a comprehensive set of model analysis and transformation operations in an industrial setting revealed some important still existing deficiencies of the MOFLON/FUJABA model transformation language and environment.

First of all some of us are not convinced that the usage of a visual notation has significant advantages compared to a textual notation as supported by other declarative model transformation languages. A textual notation is more compact, simplifies all kinds of version and configuration management tasks and does not force its users to spend hours beautifying the layout of huge diagrams. Therefore, we are planning to develop an alternative concrete textual syntax for MOF 2.0 class diagrams and SDM graph transformation diagrams as a new front-end for Fujaba/MOFLON.

Furthermore, we have made the experience again and again that graph transformation rules are very useful for the specification of structural model properties and transformations, but shouldn't or even can't be used for purposes such as regular expression checking, complex mathematical computations, or navigation along complex paths through a network of linked objects. As a consequence, Fujaba/MOFLON already offers means to combine SDM diagrams with pure Java code, OCL expressions, and a small proprietary path expression language. All these three extensions have serious draw-backs: OCL and path expressions are not expressive enough, whereas programming in Java requires intimite knowledge of the environment's code generator backend. Therefore, we are just designing a Java-like action language for Fujaba that extends the Java programming language conservatively with a small number of urgently needed constructs and hides any details concerning e.g. the design of the API of our model repository.

Finally, we are still looking for more appropriate solutions how to write reusable pieces of graph queries and transformations. Right now concepts are missing for the definition of generic queries or transformations that are parametrizable with names of attributes, associations (association ends), and classes. Furthermore, support for the definition of user-defined constructors and destructors is missing that are automatically called when graph transformation rules create or destroy objects. In this way, it would be possible to encapsulate handling of auxilary objects and object properties at well-defined places in a graph transformation specification.

There are further issues that cannot be explained here due to lack of space which may be summarized as follows: declarative graph/model transformation languages support — compared to standard imperative programming or scripting languages — the specification of model analysis and transformation operations on a considerably higher level of abstraction. But all graph transformation languages we are aware of have quite a number of deficiencies as explained above. They still need careful fine-tuning of their design (but not the development of yet another completely new transformation language). Otherwise, industry will still continue to use existing imperative (object-oriented) languages to solve their model analysis and transformation problems in the future.

References

[FR07] T. Farkas and H. Röbig. Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses. In B. Rumpe M. Conrad, H. Giese and B. Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III*, number 2007-01 in Informatik Bericht TU Braunschweig. Institut für Software Systems Engineering, Technische Universität Braunschweig, Germany, 2007. in German.

[Fuj] Fujaba Homepage. http://www.fujaba.de.

[JKS06] J. Jakob, A. Königs, and A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In A. Corradini, editor, *International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science (LNCS)*, pages 321–335, Heidelberg, 2006. Springer Verlag.

[MAA] MAAB Homepage. http://www.mathworks.com/industries/auto/maab.html.

[MAT] MATLAB Homepage. http://www.mathworks.com/products/.

[Min] Mint Homepage. http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint.

[MOF] MOFLON Homepage. http://www.moflon.org.

[MSD06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199, pages 200–214. Springer-Verlag, October 2006.

[NKS+05] Sandeep Neema, Zsolt Kalmar, Feng Shi, Attila Vizhanyo, and Gabor Karsai. A visually-specified code generator for simulink/stateflow. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 275–277, Washington, DC, USA, 2005. IEEE Computer Society.

[SCFD06] I. Stürmer, M. Conrad, I. Fey, and H. Dörr. Experiences with Model and Autocode Reviews in Model-based Software Development. In C. Salzmann M. Rappl, A. Pretschner and T. Stauner, editors, *Proc. of 3rd Intl. ICSE Workshop on Software Engineering for Automotive Systems (SEAS 2006)*. ACM Press, 2006.

[SDG+07] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A.Schürr, and A. Zündorf. Das MATE Projekt-visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen. In B. Rumpe M. Conrad, H. Giese and B. Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III*, number 2007-01 in Informatik Bericht TU Braunschweig. Institut für Software Systems Engineering, Technische Universität Braunschweig, Germany, 2007. in German.

[Soh06] Michael Sohn. Korrektheitsbegriffe für modellbasierte Codegeneratoren. Master's thesis, Martin Luther University of Halle-Wittenberg, June 2006.