

Generic and Reflective Graph Transformations for the Checking and Enforcement of Modeling Guidelines

Carsten Amelunxen, Elodie Legros, Andy Schürr
Real-Time Systems Lab
Darmstadt University of Technology
{amelunx|legros|schuerr}@es.tu-darmstadt.de

Abstract

In the automotive industry, the model driven development of software, today considered as the standard paradigm, is generally based on the use of the tool MATLAB Simulink/Stateflow. To increase the quality, the reliability, and the efficiency of the models and the generated code, checking and elimination of detected guideline violations defined in huge catalogues has become an essential task in the development process. It represents such a tremendous amount of boring work that it must necessarily be automated. In the past we have shown that graph transformation tools like Fujaba/MOFLON allow for the specification of single modeling guidelines on a very high level of abstraction and that guideline checking tools can be generated from these specifications easily. Unfortunately, graph transformation languages do not offer appropriate concepts for reuse of specification fragments - a *MUST*, when we deal with hundreds of guidelines. As a consequence we present an extension of MOFLON that supports the definition of generic rewrite rules and combines them with the reflective programming mechanisms of Java and the model repository interface standard JMI.

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 does not meet the requirements of the developers and, therefore, is neglected in favor of the MathWorks Matlab Simulink/Stateow (Matlab SL/SF) [7] environment which is better adapted for specifying, designing, implementing, and checking the functionality of new control functions. In fact, Simulink (a very small example is depicted in Fig. 1) supports a block-oriented style of modeling that combines the dataflow programming paradigm with differential equation solvers, whereas Stateflow adds a discrete event and state-

oriented style of modeling.

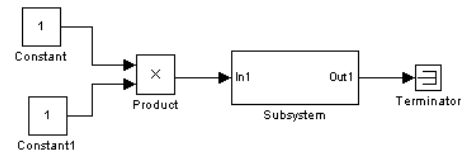


Figure 1. Matlab Simulink model

To improve the correctness and the efficiency of models and prevent typical modeling problems, generally accepted modeling guidelines such as the MathWorks Automotive Advisory Board (MAAB) catalogue [6] are usually adopted. These modeling guidelines are either manually or automatically checked during audits using tools like the Mathworks Model Advisor [7]. Nevertheless, the modeling guidelines are numerous (e.g. the MAAB catalogue contains more than 50 guidelines) and, for huge models, this can add up to a few hundreds or even thousands of violations that must be corrected manually by the modeler.

The automation of such a task would be obviously of great advantage. Nevertheless, we are not aware of any tool support in this direction except of our Matlab SL/SF Model Analysis and Transformation Environment MATE [11].

In addition to the urgent need for such a tool, another motivation for starting the MATE project was our observation of the very low level of abstraction concerning the implementation of modeling guidelines for which imperative programming languages are generally used. 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. Our experience showed us that graph transformations generally offer a significantly better support for the specification and implementation of modeling guidelines and refactorings [2]. Though, we are not completely satisfied by the current specification of some kinds of modeling guidelines and are convinced that it could be significantly improved if the currently used graph transformation language SDM was extended by some additional concepts we want to introduce in this paper.

The rest of this paper is, therefore, structured as follows. In the next section, we present the MATE project and then, in Section 3, describe the Story Driven Modeling (SDM) syntax and the overall structure of the Matlab Simulink metamodel, as well as the running example used in the following sections. We then present in Section 4 the Java Metadata Interface (JMI) that is used for code generation purpose and show how the low-level concept of the JMI interface for generic programming purposes can be lifted to the higher level of abstraction of the graph transformation language SDM. Section 5 describes our proposed enhancements for reflective model transformations followed by Section 6 which gives an overview of related approaches and Section 7 which concludes this paper with our plans for future work concerning the design and implementation of a more powerful graph transformation environment.

2 The MATE Project

The Matlab SL/SF Model Analysis and Transformation Environment MATE provides support for semi-automatic checking and enforcement of modeling guidelines as well as for design pattern instantiation, interactive model refactoring and beautifying operations. MATE is a joint project in response to the urgent need of the automotive industry for more sophisticated tool support to assist software developers using Matlab SL/SF with the maintenance and quality assurance problems of everyday life programming. Because Matlab SF/SF is used for the development of safety-critical embedded systems, model audits have become necessary steps that must be executed with rigor, but are a time-consuming and thus cost-intensive process. Spread solution to reduce the effort of reviewing is to ensure the quality of a model already during its development. In practice, catalogues of modeling guidelines are defined and the models are continuously and automatically checked according to these guidelines during the development.

Analysis as well as refactoring of Matlab SL/SF models requires full access to the model repository of Matlab, which is possible through an API written in M-Script, a proprietary script language. Due to the fact that both the used language and the tool's API evolved over many years, learning how to program reliable model checks and transformations using this approach costs time and efforts. Furthermore, model checks written in M-Script are basically programmed in an imperative style and as such neither clearly structured nor easily readable or understandable. MATE overcomes this problem 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. The specification of these graph queries and transformations is realized with the help of the Fujaba graph transformation tool and its meta-modeling add-on MOFLON [1]. A more detailed descrip-

tion 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 and may be found in [11].

3 State of the Art of Guideline Specifications

Inside MATE, guideline specifications are based on a MOF 2.0 [9] compliant metamodel of Matlab SL/SF. This metamodel acts as graph schema for the specification of graph transformation rules. The technique of graph transformations is on the one hand applied for the detection of incorrect models as well as, on the other hand, for the (semi)automatic repair of identified errors. MATE uses the visual graph transformation approach of story driven modeling (SDM) [15]. In the following we give a brief overview of this approach and its syntax followed by some examples of guideline specifications.

3.1 The Story Driven Modeling Syntax

The central idea of story driven modeling is the combination of common UML activity diagrams with graph transformations. The essential graph schema is modeled as UML/MOF class diagram. An activity diagram is used to specify the behavior of exactly one operation of a schema class by specifying the control flow concerning the execution of several graph transformation rules. Each activity diagram describes the behavior of exactly one operation, a programmed graph transformation. Differing from classical approaches, SDM graph transformation rules are not described by two separated graphs representing the left- and right-hand side of a transformation. A transformation rule is rather described by a single graph which simultaneously describes the transformation's left and right hand sides by means of annotations and different colors. Both sides' common subgraph is depicted in black without any annotations, whereas those parts of the left-hand side which are not part of the right hand side and as such deleted by the transformation are depicted in red and annotated with the annotation *destroy*. Consequently, those parts of the right-hand side that are not part of the left-hand side (and as such created by the transformation) are depicted in green and annotated as *create*.

Beside those basic concepts, there are quite a number of additional features. For instance, there are two kinds of activities: simple activities and *for each*-activities. Contrary to a simple activity which is executed at most once if there is a match, a *for each*-activity will be executed for each match that is found when the activity is initially activated. To understand how an activity must be interpreted, let us consider Fig. 2. The example shows an activity diagram which describes the behavior of the class `Guideline`'s method `do`. Due to the *for each*-activity, all graph structures consisting of the object on which the method is called (denoted by the keyword *this*)

and a linked model with a linked block are matched. For each matched structure, the activity is left via the transition with the guard *each time*.

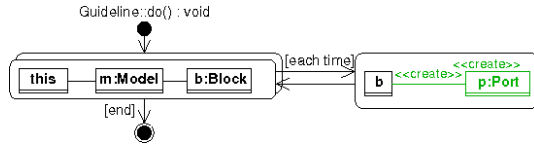


Figure 2. Simple Example of a for each-activity

Once a transformation’s object has been matched, it is denoted to be bound to its matched object. As such it can be reused by the object’s name in further activities, like e.g. the bound block object *b*. The matching of a transformation rule always starts at a given bound object. There are two different kinds of given bound objects. On the one hand, there is the bound object on which a method is called. On the other hand, each of the method’s parameter is bound from the beginning of the execution.

There are further special features of the graph transformation diagrams (e.g. optional, negative or multi-valued nodes) as well as of the activity diagrams. The complete range of available SDM constructs can be found in [15]. In the following we demonstrate how SDM can be applied for the means of guideline checking and enforcement.

3.2 Guideline Specification with SDM

In the center of SDM-based guideline specifications there is a metamodel of the particular language. In case of Matlab Simulink, the metamodel is very large and complicated; therefore, only a very small and simplified version is presented in Fig. 3. The presented simplification consists of a single package Simulink containing only the most important classes. The class *Model* represents the central class which is instantiated once per Simulink model. It contains quite a number of attributes representing all possible settings for a model. A Simulink model is composed of several functional components represented in the metamodel by the class *Block*, whose inputs and outputs are represented by the class *Port*. The several kinds of functional components (e.g. product, constant, gain) are represented by numerous subclasses of *Block*.

Many modeling guideline analyses require the execution of similar checking operations like the checking of settings or element names. Part (a) of Fig. 4 shows the story diagram of a method to verify the value of three attributes of the class *Model*. Such a method is though not suitable for the repair of the violated guideline since the failure of the story pattern means that at least one attribute is set to the wrong value without indicating which one. Therefore, it is necessary to define a separate method for each attribute that must

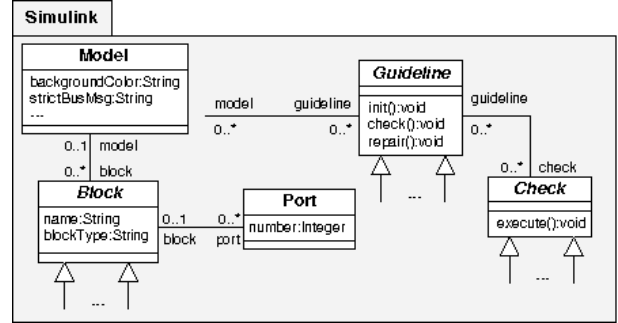


Figure 3. Very simplified Matlab metamodel

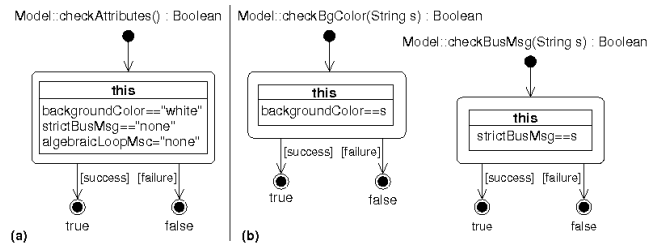


Figure 4. Check of Attribute Value

be checked as shown in part (b) of Fig. 4. This represents a repetitive and very time-consuming task considering that, for instance, the class *Model* of the Matlab metamodel contains more than 40 attributes that have to be checked in an uniform way. The corresponding checking methods differ only with respect to the used attribute name and the permitted attribute value. Thus, a generic specification of this method, in which the name and the value of the attribute that has to be checked are given as parameter value, would be of great advantage. Unfortunately, the current SDM language does not allow for the parameterization of modeling elements like classes or attributes; it only supports the parameterization with attribute values. Therefore, we have to switch to a lower level of abstraction by programming necessary generic graph analysis and transformation rules in Java against generated interfaces. On this level of abstraction it is possible to use methods for accessing attribute values, created object of a specific class, etc. with string parameters which identify the involved attribute or class (Cf. Sec. 4.1).

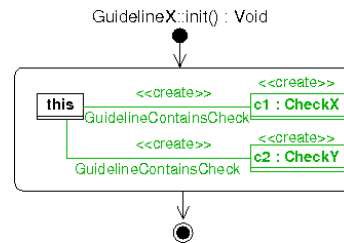


Figure 5. Initialization method for a guideline

In the simplified Simulink metamodel presented in Fig. 3,

Model is connected to the abstract class `Guideline`, whose subclasses represent modeling guidelines. The analysis of a guideline requires one or several checks which are represented by the abstract class `Check`. In fact, there are different kinds of checks which are separately defined as subclasses. A guideline check is initialized by creating the needed instances of `Check` as shown in Fig. 5, where instances of `CheckX` and `CheckY` are created and linked to the instance of `GuidelineX`. In fact, the initialization methods are very similar for all guidelines, the only difference is the name of the needed check class(es). The specification of this method is mandatory for each guideline, i.e. for a catalog containing 50 modeling guidelines such a SDM diagram must be drawn 50 times: a very time-consuming task! Here again, it would make sense to define only one generic method that can be used to initialize any modeling guideline. In general, a guideline may possess and initialize an arbitrary number of checks. Therefore, we would like to specify one method that is parameterized with a set of class names which are subclasses of `Check`. SDM in its current version does not support the definition of this kind of generic methods.

4 Generic Model Transformations

SDM graph transformation diagrams are translated into Java code that works on top of a generated graph model repository with JMI-compliant interfaces. The Java Metadata Interface (JMI) [4] standard provides powerful facilities for generic and reflective programming which cannot be used with the current syntax of the visual graph transformation. In the following, we give a brief overview of JMI before we present our proposed enhancement for generic graph transformations based on the functionalities offered by JMI.

4.1 The Java Metadata Interface

JMI defines the platform-independent Java-mapping of MOF based on the use of two kinds of interfaces: the tailored (i.e. generated) strongly typed interfaces and the reflective untyped interfaces. Tailored interfaces define methods to manipulate objects, attributes and links in a type-safe way. For this purpose, they offer specific methods for each element of a given graph schema. These interfaces inherit from the reflective interfaces which correspond to the graph schema independent part of JMI. Thus, reflective interfaces can be used on any model to provide access to its metainformation without having to know the generated interfaces.

4.2 Extension of SDM Syntax

As already mentioned, the JMI reflection layer can be used to realize generic SDM graph transformations. Although the reflective interfaces provide the same function-

ality as the tailored interfaces, there is an important difference between both. The methods of the reflective interfaces obtain all metamodel-related information as parameterized string values, whereas in case of the tailored interfaces all metamodel-related information is an integral part of the interfaces itself. Since tailored and reflective interfaces finally provide the same functionality, the parameterization of all metamodel-related information can be passed to the description of model transformations without any loss of functionality. Metamodel-related informations for which the JMI interfaces provide reflective access are attribute names, class names and association names. The parameterization of these elements in the description of model transformations provides promising possibilities. Considering the drawbacks of the current SDM graph transformations described in Sec. 3 of this paper, this functionality corresponds exactly to what is needed for the definition of generic model transformations.

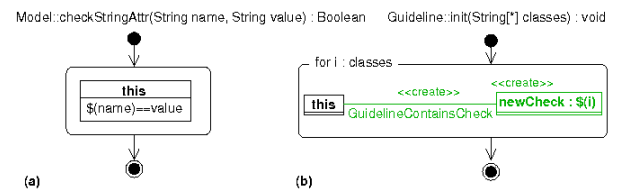


Figure 6. Generic methods

Considering the example depicted in Fig. 4, the parameterization of model information solves exactly the presented problem. As mentioned above, a usable check of more than one attribute requires the definition of as many methods as attributes are checked. Such an amount of simple checking methods leads to a number of problems. A generic check as presented in part (a) of Fig. 6 provides the perfect solution. The name of the attribute which has to be checked, `name`, is passed to the method as simple string parameter. Since this parameter contains metamodel-related information, we propose the usage of an operator represented by the symbol `$` to dereference metamodel-related information. This operator evaluates the expression which follows in conventional brackets and applies the resulting value to the metamodel depending on its particular usage. The expressions are only restricted in such a way that they must evaluate to a string value. Applying such generic transformations the before mentioned problem can be solved by a couple of generic methods as depicted in part (a) of Fig. 4, one for each needed attribute type (e.g. generic check of string attributes, generic check of boolean attributes, etc.)

The problem as mentioned in Fig. 5 can, in general, be solved with generic transformations as well, but a further problem arises. Again, metamodel-related information has to be parameterized as well, but this time more than one reflective variable has to be passed. The initialization of a guideline always includes the instantiation of several checks whose

number and types differ from guideline to guideline. Thus, a generic solution should provide a mechanism which can simply be applied with the information which checks should be instantiated. Therefore, it must be possible to create and to iterate over a set of type names.

For this purpose, we introduce a new kind of activity which is equipped with a new type of parameter, namely a typed collection¹ (e.g. *String[*]* for a collection containing String elements), and a user-defined variable as running variable for the iteration of the collection. This concept is illustrated in part (b) of Fig. 6, where our proposal is applied to a generic specification of the initialization method. The string collection *classes* contains the names of those check classes which must be created and linked to the guideline. At the top of the activity, a running variable *i* is defined as part of the expression *for i : classes*. This means that the collection after the colon is iterated; the iteration's actual object is bound to the variable. The related activity is executed for each single iteration. In case of a collection which contains objects of the metamodel rather than primitive objects the variable can be used as bound object. Finally, the specified initialization method is specified just once in the abstract class *Guideline*. Each subclass can reuse the generic initialization method by simply calling it rather than by overriding it.

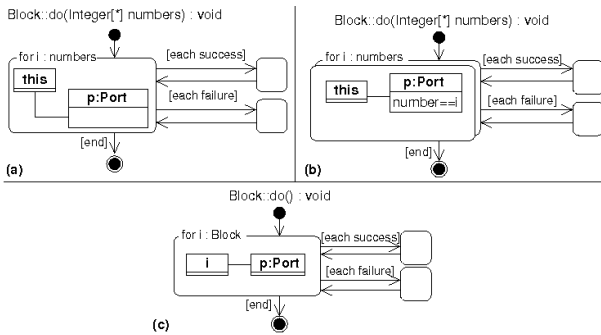


Figure 7. Iteration

The presented initialization method represents just a very simple application of an iteration, therefore, more general cases are depicted in Fig. 7. As an iteration concept which demands the processing of an iteration inside a single activity would be an unacceptable restriction, it must be possible to expand the execution of an iteration over more than one activity. Therefore, we propose to replace the old *each time*-transition with two additional transition guards as presented in part (a) of Fig. 7. The iterating activity is left via the *each success*-transition each time a match is found in the actual iteration. If there is no match found in an iteration, the activity is left via the *each failure*-transition. Similar to a *for each*-activity, the activity is left via the *end*-transition when

¹According to the used MOF 2.0 standard we use the * symbol rather than a Java-like notation.

the iteration is finished. Thus, the collection *numbers* is iterated and each time a port can be found which is linked to the block on which the method is called and has the same number as the actual iterated integer value, the execution is continued in the upper activity. If such a port cannot be matched the execution continues in the lower activity.

Part (b) of Fig. 7 combines the new set iteration construct with the existing *for each*-activity. Thus, the double box is a short-hand for an iteration with two nested loops. Therefore, for each integer element from the set *numbers* that is bound to *i*, not only one but all matches of the graph pattern enclosed by the double box are processed.

A further application of iteration is depicted in part (c) of Fig. 7. In that case, the iteration concept is used to iterate over all instances of the type² *Block*. The proposed iteration concept is meant to be applicable to collections in general which could be on the one hand a multi-valued parameter or on the other hand the set of a type's instances. The latter is denoted by using a type name instead of a collection. Again, this special feature uses a method which is provided by the reflective JMI interfaces. Since the check of a modeling guideline generally requires the checking of all of a model's elements, e.g. all the instances of *Block*, such a feature is extremely helpful.

5 Reflective Model Transformations

A further enhancement of common SDM graph/model transformation rules which is enabled by the interfaces of a generated JMI repository is the feature of reflective model transformations. Beside the interfaces for the creation, access and storage of models, JMI also provides a mechanism for the discovery of a model's metadata. Since the JMI standard is designed for MOF³ compliant metamodels, each metamodel can be described as instance of the MOF metamodel. In case of a JMI repository, this information is available at runtime, in such a way that each element of a metamodel is aware of its description as an instance of the MOF metamodel. An example of such a description at runtime is depicted in Fig. 8.

It presents a simplified cutout of a Simulink model as instantiation of a generated JMI repository for Simulink models. In the left part of Fig. 8 two instances of the class *BlockImpl* (representing a Simulink block) together with their class proxy implementation *BlockClassImpl* (representing their class as part of the Simulink metamodel) are depicted below the instance of the class *SimulinkPackageImpl* which represents the Simulink metamodel. Both of the Simulink

²Referring to the instances of a type includes all instances of all subclasses as well rather than just the instances of the class itself.

³Although JMI is originally designed for the outdated MOF 1.4, we use it in combination with MOF 2.0 which is basically a straight forward application of the standard in its current state with updated metamodel interfaces.

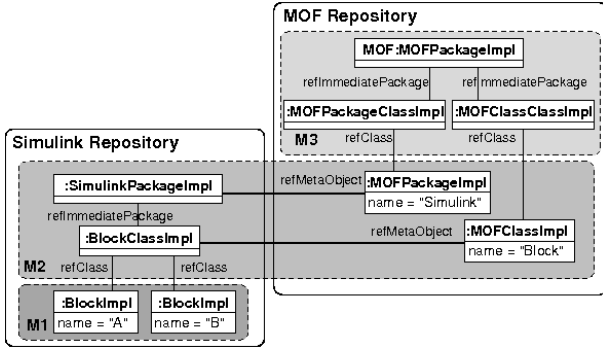


Figure 8. Relations between metalevels

metamodel’s instances, the package and the class proxy implementation, are linked to their descriptions in the MOF metamodel which is depicted in the right part. The MOF metamodel itself is generated as a JMI compliant metamodel and instantiated by the Simulink metamodel. Therefore, there are the two instances of the class MOFClassImpl (named Block) and of the class MOFPackageImpl (named Simulink, cf. Fig. 3). The first is a description of the class Block and the latter of the package Simulink. As such they are used to reflect the information which is generated into the class proxy interface and into the package interface at runtime. The suffix Impl indicates that the class implements the corresponding interface. With the information provided by the metamodel instances, in the following denoted as metaobjects, it is possible to explore and discover the Simulink metamodel without prior knowledge of its interfaces. Once a metaobject for a class, package or association is determined, the complete metamodel can be explored based on that metaobject.

The linking between a repository’s metamodel and the appropriate instances of the next higher metalevel’s metamodel allows to match a model’s metadata at runtime. Considering an object on metalevel M1, there are graph structures, on the one hand, inside the same metalevel which result from the instantiation of those associations which are connected with the object’s class. On the other hand, due to the before mentioned linking, there are also graph structures that result from a given object’s metaobject which is linked with the given object’s class. As the metaobject itself is part of a further (meta)model, there are finally metalevel-spanning graph structures. The matching of those metalevel-spanning structures can be very beneficially integrated into the existing matching of intra-metalevel structures as demonstrated in the example of Fig. 9.

The presented example matches an instance of the class Block (denoted by the object *this* since the rule is defined as operation of the class Block) and its metaobject which is an instance of the class MOFClass. The first one is part of the metalevel M1, the latter one is part of the metalevel M2. Considering Fig. 8, the rule matches one of the instances of

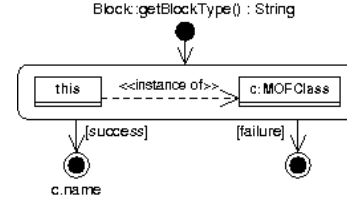


Figure 9. Simple metalevel-spanning match

class BlockImpl, either the block named A or the block named B, and the instance of MOFClassImpl, which is linked to the matched block’s class proxy implementation and named Block. Finally, the rule returns the name of the matched metaobject which represents the name of the subclass of class Block on which the operation is called. An alternative implementation without metalevel-spanning matches would demand a derivation rule for an attribute which represents the type name and a further derivation rule for this attribute in each subclass of Block. Since in practice there are a lot of subclasses, the simple rule as depicted in Fig. 9 replaces a huge amount of derivation rules.

Since the link between an object and its metaobject differs semantically from a link which connects two objects of the same metalevel, there has to be a syntactical differentiation as well. Links between objects of different metalevels are, following the UML notation, drawn as dashed arrow annotated as *instance of*. Once a metaobject is matched, all matches inside the metaobject’s metalevel are depicted in the traditional way. Beside the syntactical adjustment, there are also some restrictions that have to be considered in dealing with metaobjects. First of all, there is the given graph schema of MOF 2.0 (in the following denoted as meta graph schema) which has to be used if metaobjects are involved. This graph schema is not only immutable concerning its structure but also the metaobjects which are instances of the meta graph schema are immutable as well. The metaobjects reflect all the information about a metamodel which is transferred during code generation into the tailored interfaces of a generated repository. Thus, changing of the metaobjects would demand the generation of a new repository. Hence, metalevel-spanning graph structure can only be matched but they cannot be not modified.

The advantages of the proposed extensions are clearly visible in the example presented in Fig. 10. The example shows the specification of a method reset which resets all of a block’s attributes that have a primitive datatype to a default value.

The execution starts with the iteration over all instances of the type Block (which includes all instances of all subclasses as well) and tries to match all metaobjects which are involved in the representation of an attribute of the matched block’s effective class. This is done by matching the metaobject c of the

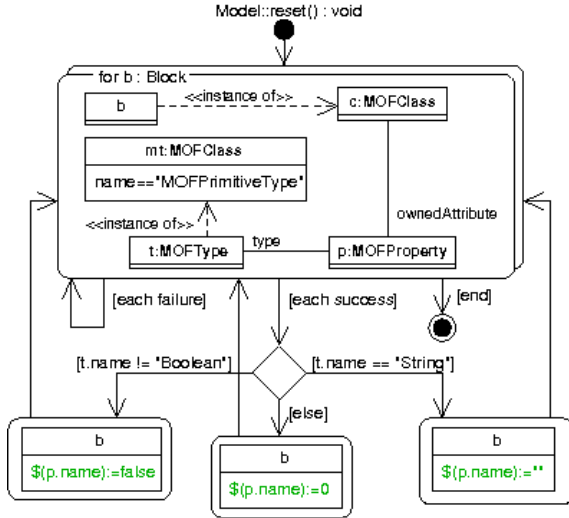


Figure 10. Reflective attribute processing

actual block *b* together with the metaobjects *p* and *t*; the first represents an attribute of the block’s effective class, the latter represents the attribute’s type. Furthermore, the metaobject *mt* of the object *t* is matched. Note that this matching is again metalevel-spanning. The access to the next higher metalevel is necessary to determine whether an attribute has a primitive type or not. Once a block and all the necessary (meta)metaobjects are matched, the name of the metaobject *t* which represents an attribute’s type can be used to distinguish which literal has to be used in the attribute assignment. The name of the metaobject *p* which represents a metaobject’s matched attribute is used as part of a generic attribute assignment.

Thus, the complete diagram specifies a kind of attribute processing with minimal⁴ prior knowledge about the concrete graph schema. Without the proposed extensions, the most convenient way to specify such a processing of attributes would be to override the class *Block*’s method *reset* in each subclass of *Block* with an implementation that is tightly coupled to the graph schema. Since there are a lot of subclasses of the class *Block*, such repeated specifications of nearly the same task would be very time-consuming. With the proposed extensions, such a rule can be specified only once and still be valid for all possible subclasses of *Block*. New subclasses or additional attributes in existing subclasses can be easily added without any modification of the presented specification.

6 Related Work

Our enhancement proposal for the graph transformation rules used by MATE must provide a support for the definition

⁴Only the class *Block* of the graph schema is used. But it could also be parameterized in a generic access.

of generic graph query and rewrite rules combined with the reflective JMI mechanisms. Generic rules have been proposed quite early for textual languages, e.g. for the Van Wijngaarden Grammars [13] that allow for the definition of potential infinite grammars with a finite number of rules, or later in [14] as successor of the Göttler’s two level grammar. The main drawback of these approaches is their complexity. The textual counterpart to the parameterization of modeling element described in Sec. 4 corresponds to the parametric polymorphism as supported e.g. by textual programming languages like Ada or Java, just to mention a few. Regarding visual graph transformation languages, the PROGRES language allow for the definition of generic rewrite rules [8], but in an limited way only. Types are first-order objects that may be used to parametrize graph transformation rules, but no means are offered to parametrize rules with attributes or associations. Furthermore, PROGRES as well as all other visual (rule-based) languages we are aware of do not offer any means for runtime reflection. The new version of SDM is the only visual rule-based language (model transformation language) we are aware of that offers both support for the visual declarative definition of generic as well as reflective transformation operations. There are also approaches [5] which provide genericity by the creation of concrete rewriting rules out of generic ones rather than by parameterized rules. Nevertheless, reflectivity is also out-of-scope. [3] proposes an interesting approach of the reflectivity in rewriting logic and its applications in several areas. One application that may be relevant for our work is the possibility of formally specifying novel reflective languages. [10] is the only tool we are aware of that offers both generic and meta transformations. Graph transformations are expressed with textual rules driven by abstract state machines. We are, however, not aware of any kind of graph transformation approach which combines visual graph transformation with reflectivity and genericity, except of [12], where first ideas how to add runtime reflection mechanisms to so-called triple graph grammars have been sketched. The basic ideas of this contribution are quite similar to the approach presented here in Section 5 due to the fact that SDM concepts have been adopted and JMI has been used as an implementation platform, too. Differences between [12] and the presented approach here include the introduction of new SDM control flow elements here as well as a more compact visual notation of generic rewrite rules.

7 Conclusion

In this paper, we propose an extension of the Fujaba/MOFLON graph transformation language SDM with constructs for runtime genericity and reflection mechanism. These extensions are motivated by our experiences with nu-

merous scalability issues encountered in an industrial case study. In this case study we were continuously faced with the problem to specify many almost identical graph transformation operations. Furthermore, we were forced to extend this set of operations, whenever the corresponding graph schema (i.e. the metamodel of the modeling language Matlab SL/SF) was modified. Unfortunately, Matlab SL/SF is an extensible language that does not make a hard distinction between a language extension and the definition of a new library of block types or the definition of a set of annotations for efficient code generation purposes. Therefore, we were spending more and more time to copy and past almost identical graph transformation operations.

As a consequence we were looking for means how to lift the available means of Java and the JMI standard (used inside Fujaba/MOFLON) for programming generic and reflective model analysis and transformation operations to the more declarative and more abstract level of SDM. We started our efforts with the implementation of needed generic and reflective operations in Java on top of JMI interfaces that (1) support the identification of regarded classes, attributes, and associations by means of string parameters and (2) offer a complete set of resources for querying metamodels of models. By adding appropriate features to SDM, we are now able to use arbitrary string parameters and expressions as class, attribute, and association names in graph transformations and to query models, metamodels and metametamodels in the same graph transformation operation visually. As a consequence a single graph transformation operation may first examine the graph schema of a given instance graph and extract some properties of the regarded language of graphs. Afterwards these properties may be used to control and instantiate generic graph transformations on the instance level.

The resulting new version of SDM is easy to use and solves all our problems with the never ending specification of very similar graph transformation rules in our industrial case study. Nevertheless, it has a number of drawbacks that should be addressed by future research activities: the underlying code generation machinery of Fujaba/MOFLON excludes the manipulation of graph schemata at runtime. Furthermore, generic operations with string parameters for metamodel element names are no longer type-safe and may throw new sorts of exceptions at runtime (such as *unknown class*). Therefore, we have to add exception handling mechanisms to SDM that were not needed as long as SDM was a strongly typed language. It is a matter of debate whether the polymorphic type systems of functional programming languages or the two-level type system of PROGRES can be adapted such that a future version of SDM combines strong typing with genericity and runtime reflection mechanism.

References

- [1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.
- [2] C. Amelunxen, E. Legros, and A. Schürr. Checking and Enforcement of Modeling Guidelines with Graph Transformations. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proceedings of the Third International Symposium on Applications of Graph Transformations with Industrial Relevance*, October 2007.
- [3] M. Clavel and J. Meseguer. Reflection in rewriting logic and its applications in the maude language. In *Proceedings of IMSA-97*, pages 128–139, Japan, 1997.
- [4] R. Dirckze. *Java Metadata Interface Specification - Version 1.0*. Unisys, June 2002. <http://java.sun.com/products/jmi/>.
- [5] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. V. Eetvelde. Shaped Generic Graph Transformation. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proceedings of the Third International Symposium on Applications of Graph Transformations with Industrial Relevance*, October 2007.
- [6] MathWorks Automotive Advisory Board Homepage. <http://www.mathworks.com/industries/auto/maab.html>.
- [7] MATLAB Simulink/Stateflow Homepage. <http://www.mathworks.com/products/>.
- [8] M. Münch. *Generic Modelling with Graph Rewriting Systems*. RWTH Aachen, November 2002. PhD Thesis.
- [9] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, March 2003. ptc/03-10-04.
- [10] OptXware Research and Development LLC. *The Viatra-I Model Transformation Framework Pattern Language Specification*, August 2006. <http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/ViatraSpecification.pdf>.
- [11] I. Stürmer, I. Kreuz, W. Schäfer, and A. Schürr. Enhanced simulink and stateflow model transformation: The MATE approach. In *Proc. of MathWorks Automotive Conference (MAC 2007)*, Dearborn (MI), USA, June 19-20 2007.
- [12] P. van Gorp, O. Muliawan, and D. Janssens. Hybrid transformation modeling: A case study based on developer interaction. In *Contribution to the 1st International Workshop on Triple Graph Grammatic*, September 2006. <http://www.es.tu-darmstadt.de/index2.php?page=2886>.
- [13] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 5:1236, 1975.
- [14] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, volume 3273, pages 290–304. Springer, 2004.
- [15] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.