Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata*

Joel Greenyer[1,**] and Jan Rieke[2,***]

[1] Politecnico di Milano, Piazza Leonardo Da Vinci, 32, 20233 Milano, Italy
greenyer@elet.polimi.it
[2] University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, Germany
jrieke@uni-paderborn.de

Abstract. Declarative model transformation languages like QVT-R and TGGs are particularly convenient because mappings between models can be specified in a rule-based way, describing how patterns in one model correspond to patterns in another. The same mapping specification can be used for different transformation and synchronization scenarios, which are important in model-based software engineering. However, even though these languages already exist for a while, they are not widely used in practice today. One reason for that is that these languages often do not provide sufficiently rich features to cope with many problems that occur in practice. We report on a complex model transformation that we have solved by TGGs. We present advanced extensions of the TGG language that we have integrated in our tool, the TGG Interpreter.

Keywords: model transformation, Triple Graph Grammar (TGG), case.

1 Introduction

Declarative model transformation languages like QVT-Relations and TGGs are particularly convenient because mappings between models can be specified in a rule-based way, describing how particular patterns in one model correspond to particular patterns in another. The same mapping specification can often be interpreted for different application scenarios, e.g., for the forward transformation from a given source model to a target model or for the backward transformation from a given target model to a source model. It can furthermore be used to keep corresponding models synchronized when changes occur to either one.

* This work was developed in the course of the Collaborative Research Center 614, Self-optimizing Concepts and Structures in Mechanical Engineering, Univ. of Paderborn, and was published on its behalf, funded by the Deutsche Forschungsgemeinschaft.
** This work was elaborated mainly while the author was working at the University of Paderborn, Germany.
*** Supported by the International Graduate School Dynamic Intelligent Systems.

A. Schürr, D. Varró, and G. Varró (Eds.): AGTIVE 2011, LNCS 7233, pp. 222–237, 2012.

However, even though these languages already exist for a while and a range of (mostly academic) tools have been developed in the past, these languages are not widely used in practice today. One reason for that is that these languages often do not provide sufficiently rich features to cope with many practical transformation problems. As a consequence, the formalisms may seem appealing at first, but many developers faced with real-life problems quickly return to "program" their transformations, using an operational language.

In this paper, we report on a complex model transformation that we have solved by TGGs (Sect. 3). We present advanced extensions of the TGG language that we have integrated in our tool, the TGG Interpreter. First, we describe the integration of OCL for describing attribute value constraints and application conditions (Sect. 4). We especially support the definition of custom operations that can be reused in the TGG rules, making them more readable.

Second, we present how constraints on stereotypes in UML domains can be conveniently specified in the TGG rules (Sect. 5). This extension is crucial because today many specific languages are defined by providing profiles for UML.

Third, we present a rule generalization concept, revising the one presented earlier by Klar et al. [15] (Sect. 6). By using generalization, we greatly reduced the number of redundant patterns that needed to be specified for our example.

Last, in our case study we experienced that there are many transformation rules where in some cases we wish to create elements in the target model, but in other cases we wish to reuse elements or whole patterns that were created in the target model by previous rule applications. We present an advanced concept for controlling the reuse of model patterns in the target domain in Sect. 7.

Furthermore, we informally discuss important properties of our TGG extensions and the transformation algorithm in Sect. 8, report on related work in Sect. 9, and conclude in Sect. 10. But first, we briefly introduce TGGs.

2 Triple Graph Grammars

Triple Graph Grammars (TGGs) [20] allow us to define sets of corresponding graphs. An element of this set is typically a triple consisting of two independent graphs that are linked via a third graph, called the *correspondence* graph. Because of this triple structure, such a graph is also called a *triple-graph*. These different graphs in a triple-graph are typed over different type graphs. TGG rules are non-deleting graph production rules that describe how, based on a start graph or *axiom*, triple-graphs can be created. Triple-graphs that can be created by a TGG are called *valid* triple-graphs.

Transferred to the "modeling world", TGGs define sets of corresponding models, also called *triple-models*, where the independent models, called *domain models*, are instances of different meta-models. The domain models are linked via a *correspondence model*, which is an instance of a correspondence meta-model.

TGGs can be interpreted for different *application scenarios*. In this paper, we focus on the *forward transformation* scenario: A model of one domain is given, called the *source* domain in the following. A TGG can now be operationalized

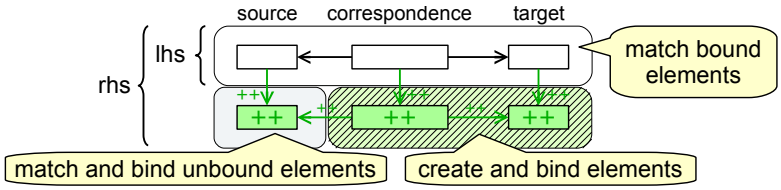

Fig. 1. The interpretation of a TGG rule for the forward transformation

to create a model of the opposite domain, called the *target* domain, and a correspondence model, such that the resulting models form a valid triple-model.

A TGG rule consists of nodes and edges that represent objects and links in the domain models. Since a TGG rule is a non-deleting graph grammar rule, the nodes and edges appear either on the left-hand side (lhs) *and* right-hand side (rhs) of the rule, or they appear on the right-hand side *only*. The former nodes and edges are also called *context nodes* and *context edges*, the latter are called *produced nodes* and *produced edges*. Context nodes are displayed as white boxes with a black border; produced nodes are displayed as green boxes with a dark green border and a "++" label. Context edges are displayed as black arrows; produced edges are displayed as dark green arrows with a "++" label.

Fig. 1 shows an abstract illustration of a TGG rule and how it is *applied* in a forward transformation scenario. Consider a state during the transformation where some rules were already applied and some elements in the source model were already translated to some target model elements. After a rule application, when an object in the model is matched by a node or created according to a node, we say that this object is *bound* to that node. A TGG rule is applied as follows: First, a match of the rule's context and source domain graph pattern must be found in the source model and the already created target and correspondence model. In this match, context nodes must be matched only to already bound objects and source produced nodes must only be matched to yet unbound objects. If such a match can be found, target and correspondence model elements can be created according to the produced target and correspondence pattern of the rule. All matched and created objects are bound to the according rule nodes. As a consequence, a model object can only be bound once to a produced node. We call this the *bind-only-once* semantics of produced nodes. Each model object in a valid triple-model is produced by exactly one produced TGG node of one TGG rule application. Thus, the bind-only-once semantics ensures that the resulting models form a valid triple-model according to the TGG.[1]

In our TGG Interpreter, we not only track which objects are bound to which nodes. We also track which links are bound to which edges. The set of all node and edge bindings after a rule application is called a *rule binding*.

[1] Most TGG transformation engines, like Moflon [1], essentially implement the same semantics. However, these tools mostly do not capture a node/object binding in an explicit data structure: an object is considered bound if there is a link from a correspondence object pointing to it.

Also constraints on attribute values and application conditions can be formulated in a TGG rule, as explained in more detail in Sect. 4. Furthermore, we have introduced the concept of *reusable nodes* and *reusable edges*, displayed in gray with a "##" label. They can be interpreted either as produced nodes and edges or as context nodes and edges [10], as explained in more detail in Sect. 7.

3 Example

The transformation example is a mapping from *Modal Sequence Diagram* (MSD) specifications to networks of *Timed Game Automata* (TGA), which is performed in order to find inconsistencies in the specification with Uppaal Tiga [3,9].

MSDs: MSDs are a formalism for specifying interactions among objects that may, must, or must not happen, proposed as a formal interpretation of UML sequence diagrams by Harel and Maoz [13]. In an MSD specification, the interaction among system and environment objects is specified in sequence diagrams where messages have a *hot* or *cold temperature*. The left of Fig. 2 shows an MSD specification with three MSDs. Hot messages are displayed as solid red arrows; cold messages are displayed as dashed blue arrows. On the top left, a collaboration diagram shows the object system, which here consists only of an environment object e:Env and a system object s:S.

In short, the semantics of an MSD specification is as follows: If in a sequence of interactions a message is sent in the system that corresponds to the first message in an MSD, an *active copy* of that MSD, or *active MSD*, is created. (We only allow a single active copy of an MSD at a time.) Upon the occurrence of further messages in the system that correspond to the subsequent messages in the MSD, the active MSD progresses. This progress is captured by the *cut*, which marks the occurred messages in the active MSD. If the cut is immediately prior to a message on its sending and receiving lifeline, this message is *enabled*. If a hot and executed message is enabled, it means that this message must eventually occur and that no message must occur that corresponds to another message in the diagram that is not currently enabled. Due to these liveness and safety requirements, there can be *inconsistencies* in an MSD specification. An MSD specification is *inconsistent* if there exists a sequence of environment events for which the system objects cannot avoid a violation of these requirements.

Timed Game Automata in Uppaal Tiga: Uppaal Tiga is an extension of the Uppaal model checker [2]. In Uppaal, a system is modeled as a network of Timed Automata (TA). Such a TA network consists of parallel automata that each consist of *locations* and *edges*. The edges in the parallel automata can be synchronized via *channels*. A *transition* in a TA network is one edge or multiple synchronized edges firing in the parallel automata. The edges can also have guard conditions and update expressions that assign values to variables. Variables and side-effect-free functions can be declared *globally* for the whole TGA network or *locally*, only visible within one automaton in the network.

In Uppaal Tiga, the Timed Automata are extended to Timed Game Automata (TGA), in which the edges can be either *controllable* or *uncontrollable*.

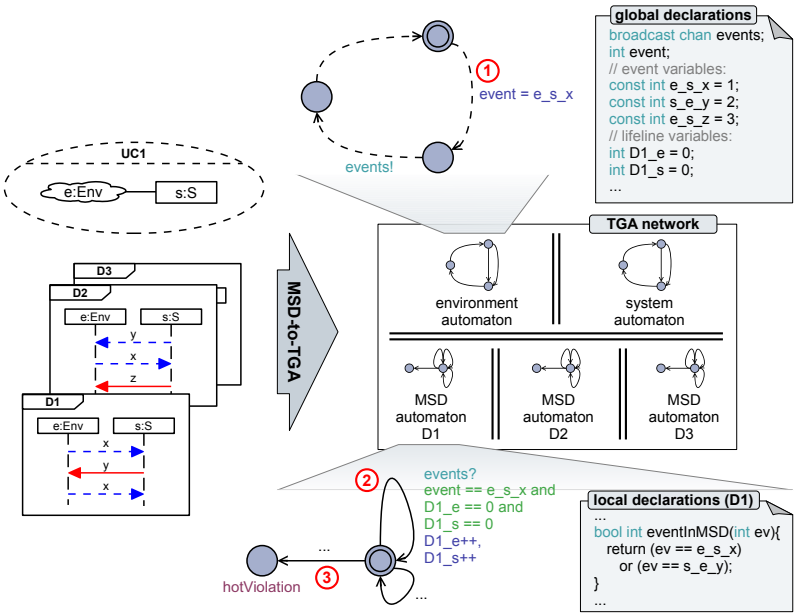

Fig. 2. MSD-to-TGA transformation overview

If only controllable edges participate in a transition, the transition is controllable by the *system*; if at least one edge is uncontrollable, the transition is controllable by the *environment*. Uppaal Tiga can check different kinds of properties in a TGA network, for example if some state is reachable by the system even though the environment will always try to keep the system from reaching that state [3].

The MSD-to-TGA Mapping: An MSD specification can be mapped to a TGA network so that Uppaal Tiga can check whether the system is always able to avoid a state that corresponds to a violation of the requirements [8,9]. If that is the case, the MSD specification is consistent, otherwise it is inconsistent.

The mapping principle is illustrated in Fig. 2: For an MSD specification, one *environment automaton* and one *system automaton* is created. For each MSD in the specification, one *MSD automaton* is created. Together, these automata form a TGA network. The environment and system automata encode the behavior of the environment and system objects sending messages in the system. The MSD automata encode the progress of the cut in the active MSD and violations that may occur in the MSD. The cut is encoded by globally declared *lifeline variables* that are created for each lifeline in each MSD.

If the environment chooses to send message `x` from object e:Env to the object s:S, this works as follows in the TGA network. First, the environment takes an edge in the environment automaton, assigning an according constant value to the variable `event` (①). Then the environment automaton takes an edge that emits over the broadcast channel `events`. This may synchronize edges in the MSD automata that represent the message. For each message in the MSD there is an

edge in the MSD automaton. This edge has a guard that ensures that it is only synchronized if the message sent is enabled in the current cut of the active MSD. It has an update label where the lifeline variables corresponding to the message's sending and receiving lifelines are increased, encoding the progress of the cut. Fig. 2 shows the edge (②) that corresponds to the first message in MSD D1.

Each message in each MSD is furthermore mapped to an integer constant declaration that represents the message in the above process. The constant name for a message has the form <name of sending object>_<name of receiving object>_<name of message>. The constant value is always the value of a previously created constant plus one. Of course, there must not be two variables or constants with the same name in the TGA specification. Thus, many diagram messages may be mapped to the same constant declaration if they have the same sender, receiver and message name. For example, the three messages called x in the MSDs D1 and D2 must all be mapped to a single declaration of the constant e_s_x. Similarly, each message in each MSD is mapped to an edge in the environment or system automaton (depending on whether it is a message sent by an environment or system object), which assigns the corresponding constant to the variable event (Fig. 2 (①)). Again, there must not be two edges that assign the same value to the event variable in the environment or MSD automaton.

For each MSD, furthermore an edge is created in the MSD automaton that is taken if a message is sent that is violating the active MSD in the current cut, i.e., the according message is not currently enabled, but nevertheless appears in the MSD. The guard and update labels of this edge (Fig. 2 (③)) are not shown in detail here. What's more important is that, in order to know whether a currently "sent" message appears in the MSD, a Boolean function eventInMSD(int ev) is created in the local declarations of each MSD automaton for each MSD in the specification. This function has a return statement that consists of a disjunction that for each message in the MSD contains a statement that renders the disjunction true if the value of the variable event corresponds to that message in the MSD. There must not be two redundant sub-expressions in the disjunction, so there is for example only one sub-expression `(ev == e_s_x)` even though the message x appears in the MSD D1 two times.

In summary, the transformation from MSD to TGA is complex for the following reasons. First, the resulting TGA models are complex, and, second, we have to distinguish several different cases when translating elements (e.g., different kinds of messages: hot and cold; messages sent from environment or system objects; messages at the beginning, middle, or end of an MSD). Third, complex string concatenations are required for variable and constant definitions, and, fourth, certain elements must not exist twice in the target model.

We realized this mapping by a TGG transformation from UML to an EMF[2]-based Uppaal Tiga model. The transformation can be downloaded as part of ScenarioTools.[3] The TGG Interpreter can also be installed separately.[4]

[2] http://www.eclipse.org/emf/

[3] http://www.cs.uni-paderborn.de/index.php?id=scenariotools

[4] http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter

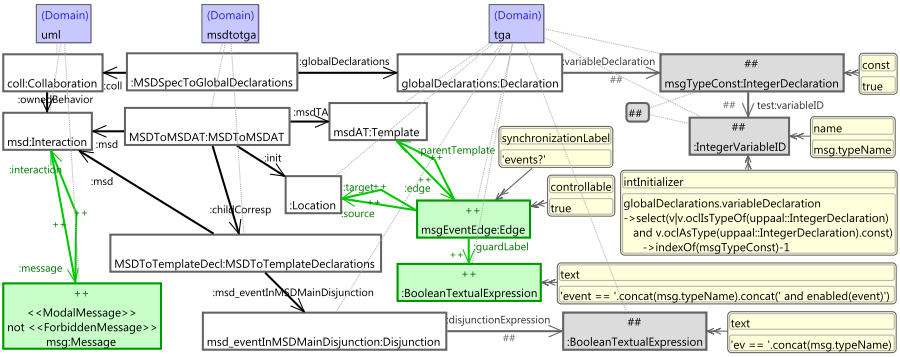

Fig. 3. Message2Edge: A TGG rule for translating messages

4 OCL Integration

This transformation is an example where many string attribute values in the target model must be concatenated from different pieces of information in the source model. In order to describe such string concatenations and other queries on the models, we have integrated OCL [18] in TGGs.

In a TGG rule, OCL expressions can be used in *attribute value constraints* and *application conditions*. They are displayed as yellow rounded rectangles in the TGG rule. In the OCL expressions, the names of nodes in the same rule can be used as variables, which are bound to the same object as the node is bound to when the rule is matched in the model (or when target model objects are created). The TGG rule in Fig. 3 shows a range of attribute value constraints.

This rule maps messages in the UML model to a range of elements in the TGA model. This diagram shows the concrete syntax of the TGG rule editor that is part of our TGG Interpreter tool suite. The domains are represented by the violet nodes at the top of the diagram that link the nodes in the domain via the thin dotted gray lines. In this rule, a message in the UML model is mapped to an edge in the MSD automaton, represented by the node msgEventEdge:Edge. This edge is added to the automaton represented by the node msdAT:Template (an automaton definition is called a *template* in Uppaal). The source and target location of the edge is the location represented by the node :Location. Furthermore, the rule maps a message to the corresponding global constant declaration and the sub-expression of the return statement in the function eventInMSD(int ev) that is declared locally for the corresponding MSD automaton (see bottom right in Fig. 2).

An attribute value constraint always points to a node, which is called its *slot node*. The top row of the attribute value constraint's rounded rectangle specifies the constrained attribute, which is called the constraint's *slot attribute*. The bottom row specifies an OCL expression, which is called the constraint's *value expression*. An attribute value constraint specifies that the slot node can

only be bound to an object if the value of the slot attribute equals the value specified by the value expression. During a forward transformation, attribute value constraints in the target domain are interpreted as assignments.

Application conditions are also displayed as yellow rounded rectangles, but they do not specify a slot node or slot attribute. They only specify an OCL expression, called the *condition expression*, which must evaluate to a Boolean value. Application conditions come in two flavors: They can be either *preconditions* or *postconditions*. Preconditions must evaluate to true in order to apply the rule. At the end of the transformation, all postconditions of all applied rules must evaluate to true.

Operationally, the attribute value constraints are considered as follows. We consider a forward transformation scenario for simplicity. The TGG engine employs a graph matching algorithm that starts with an initial matching of some source or context node in the TGG rule, and tries to find a pattern in the domain models that is isomorphic to the source and context pattern (as explained in Sect. 3). During the graph matching process, the TGG engine tries to evaluate the value expression of an attribute value constraint as soon as a candidate object for matching the slot node is found. If the result of the evaluation equals the value that the candidate object carries for the slot attribute, we say that the attribute value constraint *holds*. A node can be bound to an object if all attached attribute value constraints hold; then the graph matching can continue, otherwise the algorithm backtracks.

When evaluating the value expression, it may however happen that a variable in the expression is unbound because it corresponds to a node in the rule that is not yet bound. In this case, the attribute value constraint is marked for a delayed evaluation. It is evaluated as soon as all the nodes that appear as variables in its value expression are bound. If the constraint holds, the graph matching continues. If it turns out that the constraint does not hold, the graph matching backtracks. When backtracking, another binding for nodes that appear as variables in the constraint's value expression may be found so that the attribute value constraint holds, but the graph matching may also backtrack to find another candidate object for the constraint's slot node.

When creating elements in the target model, the attribute value constraints are interpreted as assignments. This means that when an object is created in the target model according to a node, the value expression of each attached attribute value constraint is evaluated. The value is then assigned to the slot attribute of the object created for the slot node. There may also be a delayed evaluation if the value expression of one constraint refers to a node that was not yet created.

At the end of the transformation, all attribute value constraints of all applied TGG rules are checked once again. The transformation is only correct if all attribute value constraints hold.

Preconditions are evaluated as soon as all the nodes that appear as variables in the condition expression are bound. The graph matching backtracks if the condition expression evaluates to false. Postconditions are evaluated for all rule

applications at the end of the transformation. The transformation is only correct if all postconditions are satisfied.

The above section for example mentions the naming scheme for the constant that represents a particular message in the TGA network. This name appears not only in the global declarations, but also in the update and guard labels and the local declarations of the different automata. In order to avoid that complex OCL expressions occur redundantly in the TGG rules, the TGG Interpreter allows the transformation engineer to define custom derived attributes for domain model elements within the transformation definition. These custom attributes can be defined in a separate OCL file and the OCL expressions within the TGG rules can refer to these attributes. For the MSD-to-TGA mapping, we have for example defined the custom derived attribute typeName for UML messages. It produces the string <name of sending object>_<name of receiving object>_<name of message> as explained above. This derived attribute is used three times within OCL expressions in the TGG rule shown in Fig. 3.

5 Stereotype Constraints

With the powerful UML tools that are being developed around the Eclipse implementation of UML2,[5] model-based development in practice increasingly employs UML and its lightweight profile extension mechanism [19]. We have for example used a profile to add the temperature attribute of messages in MSDs to UML sequence diagrams (similar to Harel and Maoz [12]) or to mark objects in the collaboration diagram as system or environment objects (see Fig. 2).

In transformations involving stereotyped UML models, it is crucial to be able to specify that a certain UML object has a particular stereotype applied or not. For this purpose, we have extended TGG rules by *stereotype constraints* that can be added to nodes in UML domains. Stereotype constraints specify that a node can only bind a UML object if a certain stereotype is or is not applied to that element. These constraints are shown within a node's label. An entry in double angle brackets represents a required stereotype. If preceded by the keyword not, this stereotype must not be applied.

The node msg:Message of the rule in Fig. 3 shows an example where the stereotype ModalMessage must be applied and the stereotype ForbiddenMessage must not be applied. When a UML object is created according to a node with a positive stereotype constraint, the stereotype is applied to the object.

If a positive stereotype constraint is added to a node, it is possible to add attribute value constraints where the slot attribute is an attribute defined by the stereotype that the stereotype constraint refers to. The TGG rule in Fig. 4 shows an example where the attribute value constraint added to the node specPart:Property refers to the attribute partKind, an attribute defined by the stereotype SpecificationPart.

[5] http://www.eclipse.org/uml2/

6 TGG Rule Generalization

Generalization is a powerful mechanism in object-orientation for reusing and extending existing solutions. Klar et al. have introduced this concept to TGG rules [15] and realized it within the MOFLON tool suite. In our example transformation, there are different kinds of messages that need to be mapped to the TGA model. Some elements in the TGA model must be created for all messages, some must be created e.g., only for environment messages. For this purpose, we adopted the rule generalization concept proposed by Klar et al. with some improvements.

A TGG rule describes a relation between sets of objects. Klar et al. argue that "generalization usually means that a member of a more specialized type also is a member of the more general type" and, thus, whenever a more specialized TGG rule is applicable, also the more general rule should be applicable [15, Sect. 4.1]. We call this the *guiding principle* of TGG rule generalization in the following.

To ensure that, Klar et al. define a number of syntactical constraints for a TGG rule that specializes another. These constraints require, first (1), that a specialized rule contains a copy of the general rule [15, rule 14]. Second (2), context nodes in the specialized rule may be replaced by nodes with a more specialized type [15, rule 15]. Third (3), produced nodes in the general rule can be converted to context nodes in the specialized rule [15, rule 15]. Forth (4), new nodes and edges may be added to the context and produced pattern of the rule [15, rule 16], and, fifth (5), further attribute value constraints and application conditions may be added to the rule [15, rule 17]. Last (6), the specialized rule must have a higher *priority* than the more general rule [15, rule 10]. Priorities are numbers assigned to rules; the MOFLON transformation engine will first try to apply rules with a higher number. This way it is ensured that a more general rule is only applied when any specialization of that rule cannot be applied.

In the TGG INTERPRETER, a specialized TGG rule also basically consists of a copy of the more general rule (as (1) above) and nodes, edges, and constraints may be added to the specialized rule (as (4) and (5) above). Furthermore, nodes may be replaced by nodes with a more special type class. In contrast to (2) above, this is *allowed* also for produced nodes, since it does not violate the guiding principle of TGG rule generalization. However, it is *not allowed* to convert produced nodes to context nodes in the more specialized rule (as (3) above). This is not allowed because, due to the bind-only-once semantics of produced nodes, this would violate the guiding principle of TGG rule generalization.

Another difference in the TGG rules of the TGG INTERPRETER tool suite compared to the MOFLON tool suite is that in order to create a specialized rule, the transformation engineer does not literally need to create a copy of the more general rule first. Instead, the rule diagram of the more specialized rule just contains the added nodes, edges, and constraints, and such nodes from the more general rule to which additional edges and constraints are attached. Also, the specialized rule contains the nodes from the more general rule which are given a more specialized type. All other nodes from the more general rule are only "copied" into the specialized rule during transformation-time. This makes the rule set better maintainable and the rule diagrams more concise.

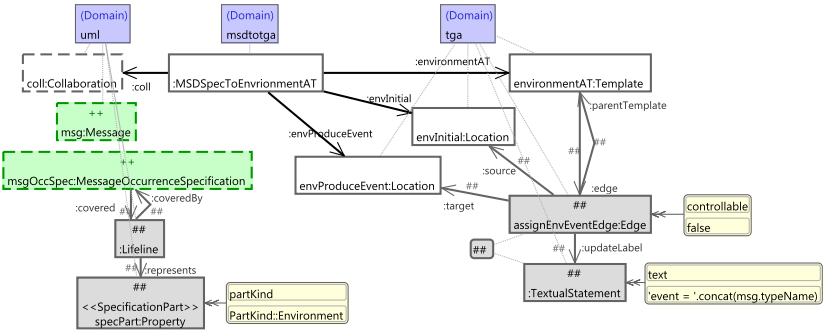

Fig. 4. EnvironmentMessage2Edge: A TGG rule for translating a minimal environment message

Nodes of a more general rule that recur in the specialized rule are called *refining nodes*. They have the same name as the node that they represent and are displayed with a dashed border. Figure 4 shows a specialization of the TGG rule in Fig. 3. This specialized rule maps an environment message (also) to an edge in the environment automaton. The nodes coll:Collaboration and msg:Message are refining nodes that appear in this rule because patterns that are added in this specialized rule are attached to these nodes. A message is an environment message if the SpecificationPart stereotype is applied to the property in the collaboration diagram that is represented by the sending lifeline of the message; moreover, the stereotype application must carry the value Environment for the partKind attribute. This is expressed by the pattern added to the UML domain of this rule. In the target domain, the message is mapped to an edge in the environment automaton with an update label as explained in Sect. 3.

Another difference to the rule generalization approach presented by Klar et al. is that we do not use priorities to ensure that the transformation engine will always try to apply more specialized rules before trying to apply more general rules. Instead of priorities we define that a more specialized rule has *precedence* over its more generalized rule. The transformation engine will not try to apply a rule if it did not try to apply another rule with precedence over that rule. The difference in this approach is that the precedence induces a *partial order* among the TGG rules whereas the priorities define a *total order*. The advantage of the precedence is that it is less restrictive and will not unnecessarily restrict the non-determinism among the rules; if we for example employ heuristics for applying TGG rules in a smart order for increasing the transformation speed, such heuristics will have more freedom to select the next rule to apply.

Note that the precedences are only relevant in the operational interpretation of the rules, i.e., they are only a directive for the transformation engine to try to apply certain rules first in a particular application scenario. By contrast, the valid triple-models are defined as those that can be produced by the TGG rules regardless of the precedences.

7 Reusable Patterns

As described in Sect. 3, each diagram message in each MSD is mapped to an integer constant declaration. The "same" diagram message can appear several times in several MSDs and must be mapped to the same constant declaration. To handle the case where yet no constant declaration exists for a message, we would need a rule where this constant declaration is represented by produced nodes. To handle the case where the constant declaration for a message already exists, we would need a rule where the constant declaration is represented by context nodes. The previous rule cannot be used for this case because of the bind-only-once semantics of the produced nodes. If there are many different elements that may or may not already exist, this leads to a large number of rules that must be created for mapping the same element. For that reason, we have introduced the concept of *reusable nodes* and *reusable edges* to TGGs [10]. The semantics of a rule with a reusable node is equivalent to two rules where the node is a produced node in one rule and a context node in the other. A transformation engine may therefore nondeterministically decide to interpret a reusable node as a produced node or as a context node. The same holds for reusable edges.

Reusable nodes can also appear in the source domain. The nodes representing the lifeline and the property in Fig. 4 are reusable nodes because they may or may not have been bound previously.

In the target domain, it is sometimes crucial to force the transformation engine to reuse a certain object structure, i.e., interpret the reusable nodes as context nodes. This is typically the case if creating another object structure instead of reusing one creates an invalid or inappropriate model. In the above case, there must for example not be two constant declarations with the same name. Furthermore, once an edge is created for an environment message in the environment automaton (see Fig. 4), this edge should be reused, because there should not be two edges from and to the same locations with identical guard, update and synchronization labels. Such constraints are sometimes part of a domain meta-model; sometimes they are only formulated for the purpose of a transformation. We call these constraints *global constraints* [10] and define that a triple-model produced by a TGG is only valid if all global constraints are satisfied. At the end of a transformation, our TGG Interpreter validates the constraints formulated in the domain meta-models and the transformation-specific global constraints that can be formulated via OCL in an external file.

If the global constraints are not satisfied at the end of a transformation, this means that the transformation engine may have to backtrack over the rule applications, reusing existing objects where previously it created them or creating new objects where it previously reused others. The latter could be required if global constraints formulate a lower bound, for example that there must be always at least two objects with certain properties in a model.

The TGG Interpreter, however, currently cannot backtrack over rule applications. Since in most cases the global constraints formulate upper bounds, such as that there must be only one object with certain properties in the model, it is in most cases sufficient to try reusing objects wherever this is possible.

The TGG Interpreter therefore implements a *reuse-before-create* semantics for reusable nodes. This is similar to the check-before-enforce semantics in QVT-Relations [17, Sect. 7.2.3]. In contrast to QVT-Relations, however, the reuse-before-create semantics is only *one possible operational interpretation* of reusable nodes in TGGs—it is *not* part of the general TGG semantics.

The reuse of an object or a link is decided for each reusable node and edge. Sometimes, however, this could lead to unintended effects. Consider the two reusable nodes assignEnvEventEdge:Edge and :TextualStatement in the TGG rule of Fig. 4. In a case where some environment messages were previously translated, there would be an uncontrollable edge in the environment automaton that the reusable node assignEnvEventEdge:Edge could always reuse. However, the update label statement attached to that edge may not be reusable, because the edge does not correspond to the "same" message. In this case, a second update label statement would be attached to the same edge, which is not what we intended.

As a solution, we have introduced the concept of *reusable patterns*. A reusable pattern is a set of reusable nodes and edges in a rule. It is represented by a small gray node with a "##" label that is connected to reusable nodes. The reusable pattern consists of all the connected nodes the reusable edges between them.

The semantics of a TGG rule with a reusable pattern is equivalent to two rules where all the nodes and edges in the pattern are produced nodes and edges in one rule and all the nodes and edges in the pattern are context nodes and edges in the other rule. Operationally, the TGG Interpreter will first try to reuse the pattern structure and will only try to create it if that is not possible.

8 Properties of the TGG Extensions

As described in Sect. 2 and 4, a triple-model is valid according to a TGG if (a) it can be produced by a sequence of TGG rule applications, (b) all postconditions and attribute value constraints hold for each applied TGG rule, and (c) all global constraint hold. If after a transformation all model domain elements are bound, the bind-only-once semantics and the final checking of above-mentioned constraints ensure that only valid triple-models are effectively created by the TGG Interpreter. This ensures the *correctness* of the results.

Also note that the precedence concept introduced for the rule generalization does not violate the correctness of a transformation. Intuitively, this is because the precedences are not considered when applying rules to produce the valid triple-models. Therefore, if a rule is applied in a forward transformation, the resulting bound triple-model always could have been created by a sequence of TGG rule applications that create all parts of the triple-model in parallel.

One general issue when operationally interpreting TGG rule in transformation scenarios is that at several steps during the transformation, different choices can be made on applying rules. This non-determinism leads to the problem that certain sequences of rule applications lead to producing a valid triple-model, but others do not. Our TGG Interpreter currently does not support backtracking over rule applications. Therefore, in some cases, we may not find a valid transformation result if one exists, i.e., our transformation algorithm is not *complete*.

Reusable nodes and rule inheritance potentially increase the variety of choices that the transformation engine has during a transformation. We plan to implement a backtracking mechanism in our TGG Interpreter. The backtracking mechanism should especially be able to consider choosing a more general rule instead of selecting only the most special ones applicable. It should also be able to change the interpretation for a reusable node or pattern (interpreting it as a produced node/pattern instead of a context node/pattern or vice versa).

As mentioned above, it may happen that several valid triple-models can be created from a source model, in which case the transformation result is not unique. We currently support no analysis methods that help to determine whether a transformation result is unique. Hermann et al. [14] present an approach that uses critical pair analysis to determine whether the transformation result of a TGG may not be unique.

9 Related Work

Dang and Gogolla [4] presented an approach for using OCL for specifying attribute value constraints and application conditions within TGGs. In their approach, they specify TGG rules textually, including a number of OCL statements. Then an OCL framework can execute the TGG rules, including the assignment of attribute values in the target domain. Compared to the approach presented here, however, they cannot define custom derived attributes for domain elements.

Golas et al. [7] show how to integrate application conditions in TGGs. They extend a formal framework for TGGs to show the termination, information preservation, correctness and completeness of transformations based on the extended TGGs. Their application conditions are restricted to formulating constraints on the context part of the rule. Also they assume that constraints in the source model are only evaluated in the scope of the already bound part of the source model. In the TGG Interpreter, constraints are evaluated with respect to the whole source model. We plan to investigate restrictions to our constraints that are necessary to ensure the completeness and information preservation of our transformations. Klar et al. [16] show that efficient translators for TGG with NACs are still preserving the fundamental TGG properties. However, these NACs are restricted to forbidding the existence of model elements.

To the best of our knowledge, there are no other TGG or QVT engines which provide a convenient support for constraints on stereotypes in UML models. Giese et al. [6] present a TGG-based transformation of a UML model with a SysML profile, but no indication is given on if and how constraints on stereotype applications are supported by their transformation engine.

Besides TGGs in Moflon, we are not aware of another relational transformation engine supporting a rule generalization concept. The comparison with other, non-relational model transformation languages is beyond the scope of this paper. We refer to Wimmer et al. [21], who compare the TGG rule generalization

concept of Klar et al. to the rule generalization concept of ATL[6] and ETL.[7] Guerra et al. [11] present a technique to specify transformations declaratively by relations between models that must or must not exist. Similar to rule generalization, it is a promising approach to make transformation specifications more intuitive. They also support attribute constraints.

Geiger et al. present a TGG engine [5] in which produced nodes can be matched multiple times to target objects. This violates the bind-only-once semantics for produced nodes, which is crucial for creating valid triple-models.

10 Conclusion and Outlook

In this paper, we reported on practically relevant TGG extensions that we elaborated and implemented in the TGG Interpreter. We extended TGGs by OCL for specifying attribute value constraints, application conditions, and custom attributes. We also integrated support for specifying constraints on stereotype applications and elaborated a rule generalization concept, refining the ideas of Klar et al. [15]. Last, we extended the concept of reusable nodes to reusable patterns to better control the reuse of model structures in target models.

With these extensions, TGGs become a powerful and flexible formalism for solving many complex model transformation and synchronization problems. We used these extensions in different transformation scenarios. Especially, the rule generalization improves the maintainability of the rule set. Complex OCL constraints and conditions also frequently occur in practical transformations.

We have also identified some open challenges. For example, using rule generalization in our example, we still ended up with some redundant rule patterns. We are therefore planning a more flexible rule extension mechanism. Furthermore, the reuse-before-create semantics of reusable patterns may not be practical in all cases. Therefore it could be useful to attach specific constraints on reusable patterns to more specifically control the reuse of model patterns. In addition, implementing backtracking over rule applications is also planned for the future.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
2. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL – A Tool Suite for Automatic Verification of Real-time Systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
3. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)

[6] `http://www.eclipse.org/atl`
[7] `http://www.eclipse.org/gmt/epsilon/doc/etl/`

4. Dang, D.-H., Gogolla, M.: On Integrating OCL and Triple Graph Grammars. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 124–137. Springer, Heidelberg (2009)
5. Geiger, N., Grusie, B., Koch, A., Zündorf, A.: Yet another TGG engine? In: Norbisrath, U., Jubeh, R. (eds.) Int. Fujaba Days. Kasseler Informatikschriften (2011)
6. Giese, H., Hildebrandt, S., Neumann, S.: Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010)
7. Golas, U., Ehrig, H., Hermann, F.: Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In: Rachid Echahed, A.H., Mosbah, M. (eds.) Int. Workshop on Graph Computation Models. Electronic Communications of the EASST, vol. 39 (2011)
8. Greenyer, J.: Synthesizing modal sequence diagram specifications with Uppaal-Tiga. Tech. Rep. tr-ri-10-310, University of Paderborn (2010)
9. Greenyer, J.: Scenario-Based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (2011)
10. Greenyer, J., Kindler, E.: Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling 9(1), 21–46 (2010)
11. Guerra, E., de Lara, J., Orejas, F.: Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In: Paige, R. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 83–99. Springer, Heidelberg (2009)
12. Harel, D., Kleinbort, A., Maoz, S.: S2A: A Compiler for Multi-modal UML Sequence Diagrams. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 121–124. Springer, Heidelberg (2007)
13. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling 7(2), 237–252 (2008)
14. Hermann, F., Ehrig, H., Orejas, F., Golas, U.: Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 155–170. Springer, Heidelberg (2010)
15. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC-FSE 2007, pp. 285–294. ACM, New York (2007)
16. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
17. Object Management Group (OMG): MOF Query/View/Transformation (QVT) 1.1 Specification, OMG document `formal/2011-01-01`
18. Object Management Group (OMG): Object Constraint Language (OCL 2.2) specification, OMG document `formal/2010-02-01`
19. Object Management Group (OMG): UML 2.3 Superstructure Specification, OMG document `formal/2010-05-03`
20. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
21. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 31–46. Springer, Heidelberg (2011)