Acta Cybernetica 17 (2005) 339–357.

Constraint Validation Support in Visual Model Transformation Systems

László Lengyel*, Tihamér Levendovszky*, and Hassan Charaf*

Abstract

Model-Driven Architecture (MDA) standardized by OMG facilitates to separate the platform independent part and the platform specific part of a system model. Due to this separation Platform-Independent Model (PIM) can be reused across several implementation platforms of the system. Platform-Specific Model (PSM) is ideally generated automatically from PIM via model transformation steps. Because of the appearance of high level languages, object-oriented technologies and CASE tools, metamodeling becomes more and more important. Metamodeling is one of the most central techniques both in design of visual languages, and reuse existing domains by extending the metamodel level. The creation of model compliers on a metamodeling basis is illustrated by a software package called Visual Modeling and Transformation System (VMTS), which is an n-layer multipurpose modeling and metamodel-based transformation system. VMTS is able to realize an MDA model compiler. This paper (i) addresses the relationship between the constraints enlisted in metamodel-based rewriting rules and the pre- and postconditions, (ii) it introduces the concepts of general validation, general preservation and general guarantee, which facilitate that if a transformation step is specified adequately with the help of constraints, and the step has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the transformation step refined with the constraints.

An illustrative case study based on constraint specification in rewriting rules is also provided.

1 Introduction

OMG's Model Driven Architecture [17] offers a standardized framework to separate the essential, platform independent information from the platform dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), and one or more platform-specific models (PSM) and complete implementations, one on each platform that the application

*Budapest University of Technology and Economics, 1111 Budapest, Goldmann György tér 3., Hungary, e-mails: lengyel@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

developer decides to support. The platform independent artifacts are mainly UML and other software models containing enough specification to generate the platform dependent artifacts automatically by so-called model compilers. Hence software model transformation provides a basis for model compilers, which plays central role in the MDA architecture.

Model transformation means converting an input model that is available at the beginning of the transformation process to an output model. MDA sets out a more restrictive definition: the output model should describe the same system as the input model. But our approach (VMTS [12] [25]) has been designed to be able to specify more general transformations. Model compilers can support properties to guarantee, preserve or validate them, and the presented approach is a practical application of these mechanisms. Models can be considered special graphs; simply contain nodes and edges between them. This mathematical background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models. The steps of graph transformation are rewriting rules, each rewriting rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Previous work [13] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements. It means that an instantiation of the LHS must be found in the graph to which the rule is applied (host graph) instead of the isomorphic subgraph of the LHS. Hence the LHS and RHS graphs are the metamodels of the graphs which we search and replace in the host graph.

Often it is not enough to match graphs based on the topological information only, we want to restrict the desired match by other properties, e.g. we want to match a subgraph with a node, which has a special property or which has a unique relation between the properties of the matched nodes. For example we want to match a state in a statechart model which has at least one incoming and two outgoing transitions. The metamodel-based specification of the rules [13] allows assigning OCL [18] constraints to the rules, using the guidelines of the UML standard [19]. Because these constraints are bound to the rules, they are able to express local constraints. This is inherently a local construct, because the elements not appearing in LHS or RHS cannot be directly included in the OCL statements. Although the specification has this local-nature, it does not mean that validating them does not involve checking other model elements in the input model: constraint propagation needs to be taken into account by both the algorithmic background and the user of the transformation on specifying the constraint. OCL constraints which are enlisted in the LHS and RHS graphs affect the matched instances of the LHS and RHS graphs.

We have pre- and postconditions and OCL constraints assigned to the rewriting rules. In this paper we introduce the relation between them and the applicability of this concept based on a case study.

2 Backgrounds and Related work

The purpose of contracts [16] is to help us build better software by organizing the communication between software elements through specifying the mutual obligations. Contracts are used to guarantee that these communications occur on the basis of precise specifications of what these services are going to be. For the software to be able to guarantee any kind of correctness and robustness properties, they must know the precise constraints over such communications. In a client/supplier relationship, where the client needs a certain service and the supplier provides that service, to impose certain obligations on the client as to the kind of original program state that is permissible, when the client calls the supplier or the kind of arguments that the client routine passes to the supplier. These are preconditions, and they are obligations for the client. In the other direction, we are going to express the conditions that the supplier routine must guarantee to the client on completion of the supplier's task. That is the postcondition of the contract, specifically, the postcondition of that particular routine. The postcondition is also an obligation for the supplier. Besides the pre- and postcondition the third fundamental element of contracts is the invariant. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy.

The Object Constraint Language [18] is a formal language for analysis and design of software systems. It is a subset of the industry standard Unified Modeling Language [19] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. There are four types of constraints: (i) An invariant is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A precondition to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. (iii) A postcondition to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A guard is a constraint that must be true before a state transition fires. Besides these, OCL can be used as a navigation language as well.

Graph rewriting [2, 7, 21] is a powerful tool for graph transformations with strong mathematical background. Originally it was developed as the natural generalization of Chomsky grammars to generate and parse visual languages. Instead of that graph language approach we will use the mechanism of the individual parsing steps, so-called rewriting rules for graph transformations. Rewriting rules are the atoms of graph transformation, which consists of a left hand side graph (LHS) and right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of the LHS in the graph to witch the rule being applied (host graph), and replacing this subgraph with RHS. Replacing means removing elements which are in the LHS but not in the RHS, and gluing elements which are in the RHS but not in the LHS. Replacing process consists of two steps: removing and gluing, this approach is the so-called double pushout (DPO) [21]. The graph transformation is defined as an ordered sequence of rewriting rules, in other

words we control the transformation process by sequencing the rewriting rules. The rewriting rule is a stereotype of the UML activity state (<<Rewriting Rule>>) [13].

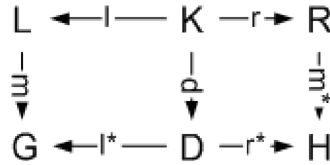

Figure 1: An illustration of direct derivation: L, K, R are the left-hand side, the interface and the right hand side graph; G and H are the graphs before and after the rule firing and D corresponds to K; l, r, l*, r*, m, d, m* are inclusions

The DPO approach accomplishes the rule firing by two steps: after finding a redex (the part of the host graph parsed by the rewriting rule), the first step removes the elements (vertices and edges) from the redex which are in the redex, but not in the RHS graph. This modified redex is referred to as interface graph. Then as a second step the elements of the RHS graph not in the interface graph but in the RHS graph are glued to the interface graph. The rewriting rule is characterized by a double pushout. The application of the rules results in a direct derivation of the host graph (Fig. 1). The category theory framework provides more flexible and more general background, so the DPO approach can be applied to many graph-like categories. For labeled and directed graphs the existence of the pushout (which is the condition to fire a rule) can be ensured by forcing the so-called gluing condition. The gluing condition consists of two parts. Firstly, the identification condition, which states that different vertices in the rewriting rule cannot match the same vertex in the host graph. Secondly, the dangling edge condition has to be dealt with as well: if a vertex should be deleted which is connected to an edge that is not inside the redex, the production rule cannot be fired. Unfortunately, this makes impossible to delete a connected vertex without considering its environment. Related to the DPO approach a rather tutorial like description can be found in [3, 5, 6, 4], and a more complete summary in [8, 21].

Our tool set, the Visual Modeling and Transformation System (VMTS) [13, 25] is an n-layer multipurpose modeling and metamodel-based transformation system. Using this environment, it is easy to edit metamodels, design models according to their metamodels, transform models using graph rewriting [13, 14]. It facilitates to check the metamodel constraints during the metamodel instantiation, and the rewriting rule constraints during the graph transformation process. VMTS can also be used to parse visual languages. The control structure can be thought of as a similar construct to programmed graph rewriting systems e.g. [20] and the rewriting process follows rules of the DPO direct derivation. Although the rewriting rules are specified in terms of the metamodel elements [13], on the instantiation level the rule firing mechanism is equivalent to the DPO rewriting rules.

VMTS benefits from the results of the mathematical background of formal lan-

guages, graph rewriting and results related to the metamodel-based software model transformation. It also incorporates several ideas from other existing environments [1, 10, 15], which implement the OCL, and enable constraints to be checked over models.

Model Integrated Computing (MIC) [22, 23, 24] is a model-based approach to software development, facilitating the synthesis of application programs from models created using customized, domain-specific program synthesis environments. MIC focuses on models, supports the flexible creation of modeling environments, and helps following the changes of the models. At the same time it facilitates code generation and provides tool support for turning the created models into code artifacts. Metamodeling environments and model interpreters together form the tool support for MIC. So far MIC is the only methodology, which requires metamodeling environments, model processors and provides a framework for them to cooperate to create Computer-Based Systems (CBS) in the practice. VMTS implements the ideas of MIC and it is able to realize an MDA model compiler.

The Generic Modeling Environment (GME) [9] is a metamodeling tool from which our system has borrowed several concepts of a metamodel-based modeling tool. GME on its own is not a transformation system, although the underlying MultiGraph Architecture (MGA) can be reached from the GReAT transformation system. GME supports constraint handling, it has a constraint interpreter called Constraint Manager.

The GReAT framework [11] is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts; it uses a proprietary notation and interpretation instead of instantiation between the rules expressed with meta elements and the match. The sequencing of the rewriting rules and parameter passing are similar in GReAT and in VMTS.

PROGRES [20] is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. There are rather few tools which support pre- and postconditions, but VMTS and PROGRES manage them. In PROGRES the precondition of a transaction is a query, which should never fail, applied to the input graph of the surrounding transaction. Similarly the postcondition of a transaction is a query, which should never fail applied to the output graph of the surrounding transaction. It is allowed to access the in- and out-parameters of its transaction, but does not distinguish between a before- and an after-state of referenced nodes.

3 Contributions

The design by contract principle and model transformation by graph rewriting mechanism has been successfully applied in checking software systems. Thus revealing the connection between the concepts of pre- and postconditions and the OCL constraints assigned to the rewriting rules seems a promising direction.

This section (i) introduces the relation between the pre- and postconditions and OCL constraints assigned to the rewriting rules, (ii) presents how to check models

based on this relation, and (iii) discusses a metamodel-based approach along with a simple but illustrative case study: while we give and explain our definitions and propositions, we can show their benefit immediately on a practical example.

3.1 Relation between the Pre- and Postconditions and OCL Constraints

For the unified treatment we give the basic definitions, which are mainly based on [16]:

Definition 1. (*Transformation and Finite sequence of steps*)
A *Transformation* and a *Finite sequence of steps* consist of n number of rewriting rules (where $n > 0$) in an ordered sequence. This sequence defines the execution order of the contained rewriting rules. The difference between a *transformation* and a *finite sequence of steps* is that a *finite sequence of steps* is always comes to an end, it always terminates. A *transformation*, however, can contain infinite number of steps.

Definition 2. (*Precondition and Postcondition*)
A *precondition* (*postcondition*) assigned to a rewriting rule is a boolean expression that must be true at the moment when the rewriting rule is fired (after the completion of a rewriting rule). If a *precondition* of a rewriting rule is not true then the rewriting rule fails without being fired. If a *postcondition* of a rewriting rule is not true after the execution of the rewriting rule, the rewriting rule fails.

Definition 3. (*Validation*, *Preservation*, *Guarantee*)
Validation of a property: a transformation step S *validates* a property P specified by a boolean expression, when the following condition always holds: if a property P was true before the step S it remains true after the execution of the step S, and if P is false, the step S fails.
Preservation of a property: a transformation step S *preserves* a property P specified by a boolean expression, when the following condition always holds: if a property P was false (true) before the step S it remains false (true) after the execution of the step S.
Guarantee of a property: a transformation step S *guarantees* a property P specified by a boolean expression, when the following condition always holds: if a property P was true before the step S it remains true after the execution of the step S, and if P is false, the step S changes property P to true.

Definition 4. (*Pre value*)
If an OCL constraint is specified in the RHS of a transformation step, $x@pre$ means the value of x immediately before the rule was fired even if the value of x has not changed.

A direct corollary of Definition 2 that an OCL expression in LHS is a precondition to the rewriting rule, and an OCL expression in RHS is a postcondition to the

	property P before the step S	**property P after the step S**
Validation	true	true
	false	step S fails
Preservation	true	true
	false	false
Guarantee	true	true
	false	true

Table 1: Truth table of the validation, preservation and guarantee properties

rewriting rule. A rewriting rule can be fired if and only if all conditions enlisted in LHS are true. Also, if a rewriting rule finished successfully then all conditions enlisted in RHS must be true.

Table 1 illustrates the truth table of the validation, preservation and guarantee properties.

3.2 A Case Study

To show the applicability and the practical relevance of the results, a case study is provided. The case study contains a rewriting rule which uses a statechart model as input model and builds a CodeDOM [25] tree (an abstract syntax like graph representation of the code to be generated) in the VMTS database which we can use easily to generate source code from it on optional language (e.g. C++, C# or J#).

In Fig. 2 there is a statechart model of a water tank. Our case study uses this statechart model as input graph and applies a rewriting rule (Fig. 3) to it.

In VMTS it is possible that the LHS and the RHS of a rewriting rule have different metamodel. In the rewriting rule depicted in Fig. 3 the metamodel of the LHS is the Statechart metamodel [19, 25] and the metamodel of the RHS is the CodeDOM metamodel [25]. On the LHS of the rewriting rule there are two states, whose meta type is statechart state, and there is a transition between them with a 0..* multiplicity on the side of the target state. It means that exhaustively applying this rewriting rule to a statechart model, it will match all states with their target adjacent states. The rule has to match the accessible adjacent states because we need them to generate the state-transitions in the source code. Obviously it is possible that a state has no outgoing transitions, and the reason why we enable the 0 in the multiplicity is that we want to match states having only incoming transitions to generate CodeDOM tree for it. On the RHS of the rewriting rule the $CTypeDeclaration$ represents a type declaration for a class, structure, interface or enumeration. $CMemberField$ can be used to denote the declaration for a field of a type, and $CMemberMethod$ to phrase the declaration for a method. $CParameter$ represents a parameter declaration for a method, property, or constructor, and $CSnippetStatement$ means a statement using a literal code fragment.

In a rewriting rule we can connect the LHS elements to the RHS elements, this

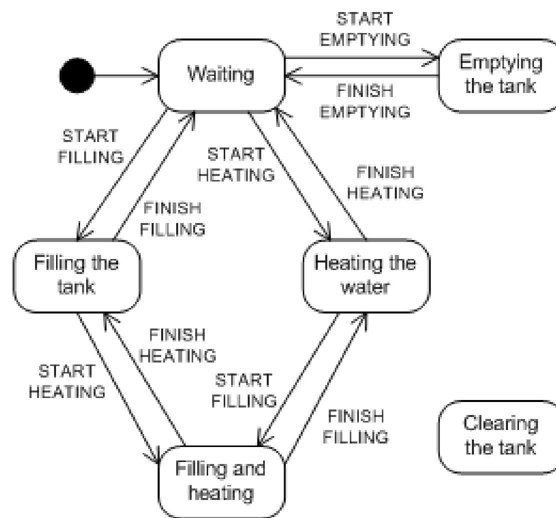

Figure 2: Case study: statechart of the water tank

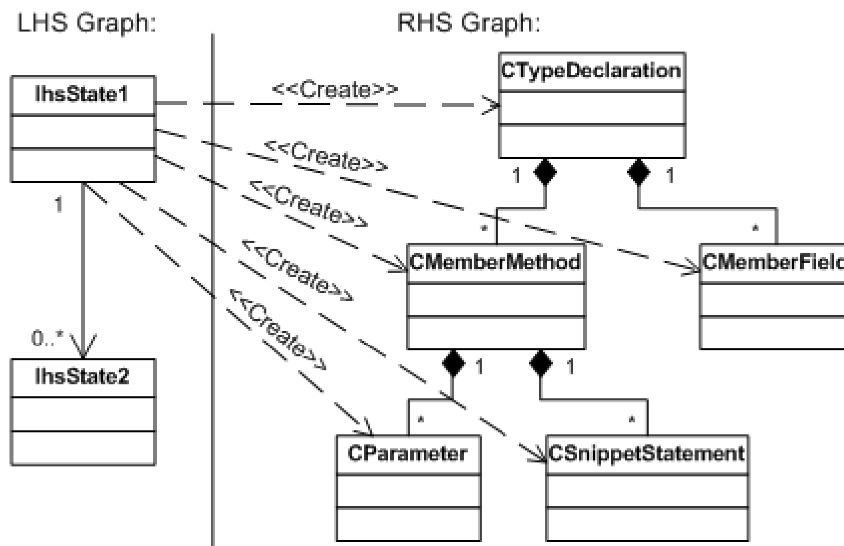

Figure 3: Rewriting rule of the case study

relation between LHS and RHS elements is called causality [11], which facilitates to assign an operation to this connection. Causalities can express either modification or removal of an LHS element, or creation of an RHS element. In Fig. 3 the causalities are denoted as dashed lines. The create operation and attribute transformation that are one of the most important part of the rewriting process are accomplished by XSL scripts. XSL scripts can access the attributes of the objects matched to LHS elements, and produce a set of attributes for the RHS element to which the causality point.

After the rewriting process the output graph has two parts. The difference between the two parts is that the first part is not affected, while the second part is affected by the rewriting rule. Accordingly we can examine the constraints contained by our rewriting rule only in the second part of the output graph, which is affected by the rewriting rule. Our case study is a fortunate case, because the whole right side and the whole output graph is generated. Hence it facilitates to check the local-nature constraints in the whole output graph.

In the next subsections we give our propositions for individual steps (rewriting rule), and for finite sequence of steps in along with the validation, preservation and guarantee properties, and we will illustrate their applicability based on the presented case study.

3.3 General Validation

Based on the validation property we can introduce the concepts of general validation. The goal of the general validation is that if a rewriting rule (a finite sequence of steps) is specified properly with the help of validation type constraints, and the rewriting rule (the finite sequence of steps) has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the rewriting rule (the finite sequence of steps) refined with the constraints.

Proposition 1. General validation (Sufficient and necessary conditions)

(i) A step S validates a property P for an input model M if the property P is enlisted in both pre- and postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. (S^{LHS} - is the left-hand side (LHS) of the step S, and S^{RHS}- is the right-hand side (RHS) of the step S). Assuming that the property P is enlisted in both S^{LHS} and S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not validate the property P.

This is a contradiction because (1) if the property P is not true the step S cannot even be fired. (2) If the property P is true the step S can be fired. The property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, it means that the property

P is true after the execution of the step S, which is equivalent to the definition of the validation. Hence the step S validates the property P.

(ii) If a step S validates a property P for a model M without conditions in the step S for the property P, then the property P can be enlisted in S^{LHS} and S^{RHS} without changing the result of the step S for the model M.

Proof. Assuming that (a) step S validates a property P for a model M without conditions in the step S for the property P, (b) one enlists the property P in S^{LHS} and S^{RHS}, then because of the newly added constraints, the result for the step S is different.

This is a contradiction because (1) if the property P is false: then the step S fails in both cases. In the first case (a) if the step S validates the property P and the property P is false, then the step S fails (definition of the validation). In the second case (b) if the property P placed in S^{LHS} and the property P is false, then the step S fails without being fired (definition of the precondition). (2) If the property P is true, the step S can be fired. In the first case (a) the property P remains true after the execution of the step S, because the step S validates the property P. In the second case (b) the property P is enlisted in S^{LHS} and S^{RHS}, which is equivalent to the definition of the validation. It means that the result can not be different.

(iii) A finite sequence of steps (S_1, $S_2 \ldots S_n$) validates a property P for an input model M if the property P is enlisted both in preconditions of the step S_1 and in the postconditions of the step S_n, and the finite sequence of steps S_1, $S_2 \ldots S_n$ has been executed successfully for the model M.

(iv) If a finite sequence of steps (S_1, $S_2 \ldots S_n$) validates a property P for an input model M without conditions in S_1^{LHS} and S_n^{RHS} for the property P, then the property P can be enlisted in S_1^{LHS} and S_n^{RHS} without changing the result of the finite sequence of steps S_1, $S_2 \ldots S_n$ for the model M.

As it is stated above, the rewriting rule of the case study generates a CodeDOM tree for the matched states. An example for the validation can be the following: the rewriting rule validates that the states with generated CodeDOM tree are not isolated (unreachable) states; it means that starting from the start state we can get at these states. To examine whether a state is unreachable, we enlist a constraint in the precondition of the rewriting rule, which checks whether the sate is isolated (Fig. 4). If the state is reachable (e.g. *Filling the tank*), the rewriting rule can be fired, but if the state is isolated (e.g. *Cleaning the tank*), the step fails (there is no proper match because of the violated constraint), as it is depicted in Fig. 5.

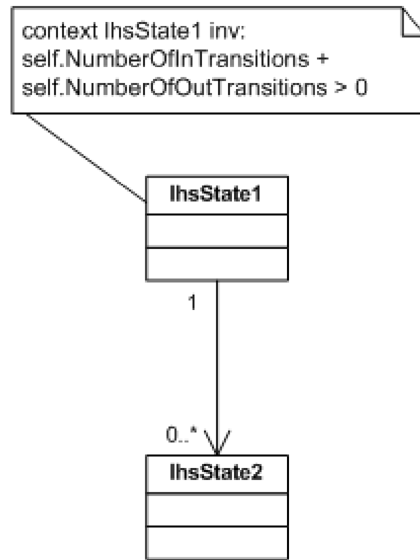

Figure 4: The LHS of the rewriting rule with an OCL constraint

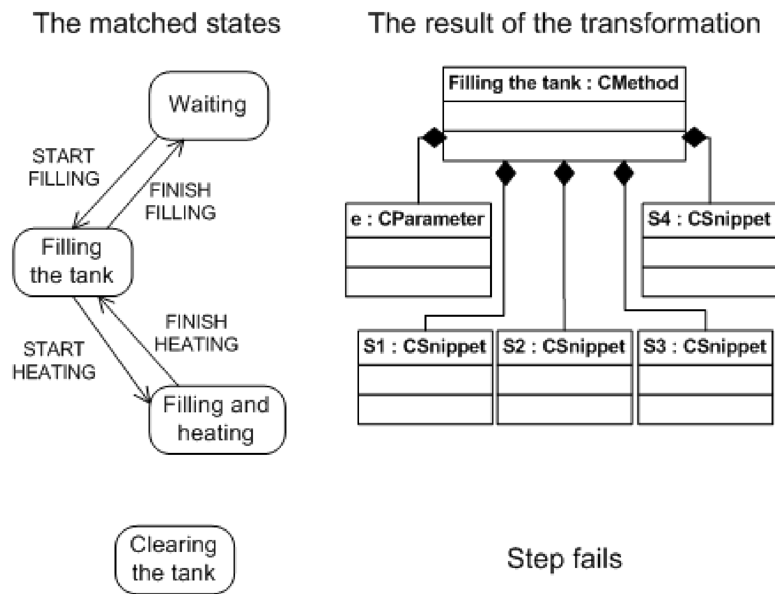

Figure 5: Examples for the matched states and the results of the transformation

3.4 General Preservation

Based on the preservation property we can introduce the concepts of general preservation. The modeler's task is to create adequate rewriting rules (finite sequence of steps) and specify them fully with constraints (pre- and postconditions). If the execution of the rewriting rule (the finite sequence of steps) finishes successfully, it preserves the required property values.

Proposition 2. General preservation (Sufficient and necessary condition)

(i) A step S preserves a property P for an input model M if the expression $(NOT\ P\ or\ P@pre)\ and\ (P\ or\ NOT P@pre)$ is enlisted as an OCL expression in the postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. Assuming that the expression $(NOT\ P\ or\ P@pre)\ and\ (P\ or\ NOT\ P@pre)$ is enlisted as an OCL expression in S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not preserve the property P.

This is a contradiction because the expression $(NOT\ P\ or\ P@pre)\ and\ (P\ or\ NOT\ P@pre)$ is true if and only if the $P@pre = P$ holds after the execution of the step S, and it means that the step S preserves the property P.

(ii) If a step S preserves a property P for a model M without a postcondition in the step S for the property P, then the expression $(NOT\ P\ or\ P@pre)\ and\ (P\ or\ NOT\ P@pre)$ as an OCL expression can be enlisted in S^{RHS}, and the step S also preserves the property P for the model M.

Proof. Assuming that (a) the step S preserves a property P for a model M without postconditions in the step S for the property P, (b) one enlists the expression $(NOT\ P\ or\ P@pre)\ and\ (P\ or\ NOT\ P@pre)$ as an OCL expression in S^{RHS}, but in this case, because of the newly added constraint, the step S does not preserve the property P for the model M.

This is a contradiction, because in the first case (a) the step S preserves the property P. In the second case (b) the enlisted constraint is equivalent to the definition of the preservation, hence the step S preserves the property P in the second case (b) as well.

(iii) A finite sequence of steps $(S_1,\ S_2\ \ldots S_n)$ preserves a property P for an input model M if the expression $(NOT\ P\ or\ P@preS1)\ and\ (P\ or\ NOT\ P@preS_1)$ is enlisted as an OCL expression in the postconditions of the step S_n, and the finite sequence of steps $S_1,\ S_2 \ldots S_n$ has been executed successfully for the model M.

(iv) If a finite sequence of steps (S_1, $S_2 \ldots S_n$) preserves a property P for an input model M without conditions in S_n^{RHS} for the property P, then the expression ($NOT\ P\ or\ P@preS_1$) and ($P\ or\ NOT\ P@preS_1$) as an OCL expression can be enlisted in S_n^{RHS}, and the finite sequence of steps S_1, $S_2 \ldots S_n$ also preserves the property P for the model M.

The preservation property is important for the individual node attributes and for the relation between certain nodes. For example if a matched node has a property called *History* with the value *Deep*, then after rewriting the generated node which corresponds to the matched node also has a *History* property with the value *Deep* (Fig. 6).

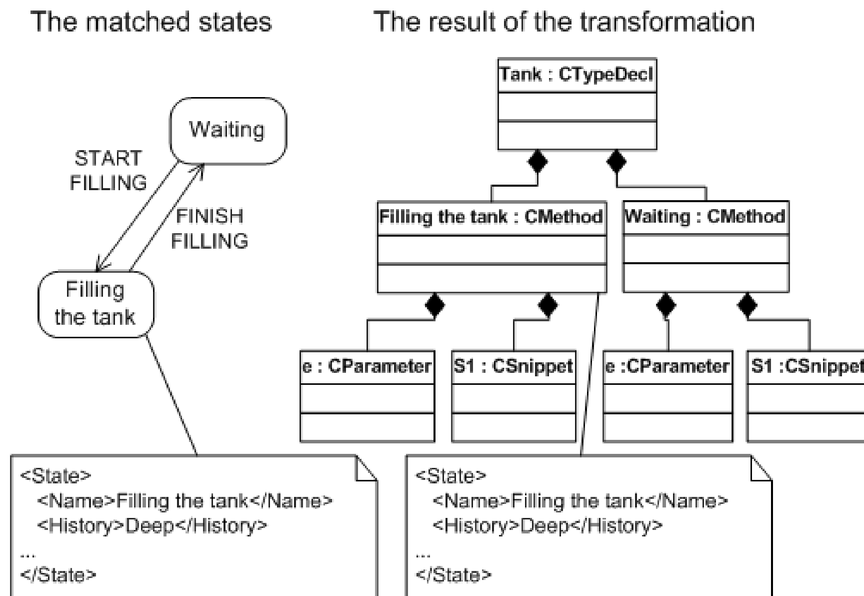

Figure 6: Example for the preservation of a property

3.5 General Guarantee

Similarly to the general validation and the general preservation, and using the guarantee property we can introduce the concepts of general guarantee.

Proposition 3. General guarantee (Sufficient and necessary condition)

(i) A step S guarantees a property P for an input model M if the property P is enlisted in the postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. Assuming that the property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not guarantee the property P.

This is a contradiction, because the property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, and it means that the property P is true after the execution of the step S, which is equivalent to the definition of the guarantee. Hence the step S guarantees the property P.

(ii) If a step S guarantees a property P for a model M without a postcondition in the step S for the property P, then the property P can be enlisted in S^{RHS} and the step S also guarantees the property P for the model M.

Proof. Assuming that (a) step S guarantees a property P for a model M without postcondition in the step S for the property P, (b) one enlists the property P in S^{RHS}, but in this case, because of the newly added constraint, the step S does not guarantee the property P for the model M.

This is a contradiction, because in the first case (a) the step S guarantees the property P. In the second case (b) the enlisted property P is equivalent to the definition of guarantee, hence the step S also guarantees the property P in the second case (b).

(iii) A finite sequence of steps (S_1, $S_2 \ldots S_n$) guarantees a property P for an input model M if (a) the property P is enlisted in the preconditions of step S_i (where $1 \leq i \leq n$), and S_i, $S_{i+1} \ldots S_n$ preserve the property P, or (b) the property P is enlisted in the postconditions of step S_i (where $1 \leq i \leq n$), moreover S_{i+1}, $S_{i+2} \ldots S_n$ preserve the property P, and the finite sequence of steps S_1, $S_2 \ldots S_n$ has been executed successfully for the model M.

(iv) If a finite sequence of steps (S_1, $S_2 \ldots S_n$) guarantees a property P for an input model M without conditions in S_n^{RHS} for the property P, then the property P can be enlisted in S_n^{RHS}, and the finite sequence of steps S_1, $S_2 \ldots S_n$ also guarantees the property P for the model M.

In the statechart model of the case study every state has a property called *IsCodeAlreadyGenerated*. If this property is true, it means that this state already has a generated CodeDOM model. Exhaustively executing our rewriting rule, it guarantees that the *IsCodeAlreadyGenerated* property of every state - which is not isolated (*Cleaning the tank*) - will be true.

3.6 Validation, Preservation and Guarantee Algorithms

After specifying a constraint, we can apply it to an individual step or to the whole transformation. The pseudo codes of the algorithms are the following:

```
VALIDATION (VMTSConstraint const, VMTSControl control,
VMTSMatch match): bool
 1 VMTSRule firstRule = control.getFirstRule()
 2 VMTSRule lastRule = control.getLastRule()
 3 ADD_PRECONDITION_TO_RULE(firstRule, const)
 4 ADD_POSTCONDITION_TO_RULE(lastRule, const)
 5 return EXECUTE_CONTROL(control, match)
```

Three parameters are passed to the validation algorithm, the first parameter is a $VMTSConstraint$ which contains the constraint we want to validate, the second one is a $VMTSControl$, which contains the rewriting rules in an ordered sequence, and the third one is a $VMTSMatch$ which contains the matched nodes. The validation algorithm selects the first and the last rules from the control, and enlists the given constraint as a precondition in the selected first step and as a postcondition in the selected last step. This will be proper if the control contains only one step, and also if the control contains a finite sequence of steps, because if there is only one step in the control, the $getFirstRule$ and the $getLastRule$ queries return the same step. Finally the algorithm calls the EXECUTE_CONTROL method and retrieves its return value.

The second pseudo code describes the EXECUTE_CONTROL method, which is used by all three (validation, preservation, guarantee) algorithms.

```
EXECUTE_CONTROL (VMTSControl control, VMTSMatch match): bool
 1 foreach VMTSRule rule of control.getRulesInOrderedSequence()
 2   if rule.hasPrecondition and not (CHECK_PRECONDITIONS(rule, match))
 3     then return false
 4   end if
 5   FIRE_RULE(rule, match)
 6   if rule.hasPostcondition and not (CHECK_POSTCONDITIONS(rule, match))
 7     then return false
 8   end if
 9 end foreach
10 return true
```

The EXECUTE_CONTROL method applies the rewriting rules contained by the given control to the passed match. If a step has a precondition, the methods calls the CHECK_PRECONDITIONS method, and if it returns false, then the execution of the step and the whole finite sequence of steps fails, and the algorithm returns false. Otherwise the method calls the FIRE_RULE function, and after that if the rule has a postcondition then the procedure is similar to the case of preconditions.

```
PRESERVATION (VMTSConstraint const, VMTSControl control,
VMTSMatch match): bool
 1 VMTSRule firstRule = control.getFirstRule()
 2 VMTSRule lastRule = control.getLastRule()
 3 ADD_POSTCONDITION_TO_RULE(lastRule, (NOT  const ||
   const@pre_firstRule) && (const || NOT  const@pre_firstRule))
 4 return EXECUTE_CONTROL(control, match)
```

The preservation algorithm selects the first and the last rules from the control, creates a constraint expression based on the given constraint: $(NOT\ P\ or\ P@preS_1)\ and\ (P\ or\ NOT\ P@preS_1)$, enlists this created expression as an OCL expression in the postconditions of the selected last step, and finally calls the EXECUTE_CONTROL method and obtains its return value.

```
GUARANTEE (VMTSConstraint const, VMTSControl control, VMTSMatch match,
int indexOfMarked): bool
 1 if (control.Rules.Length == 1 || indexOfMarked < 0)
 2   then ADD_POSTCONDITION_TO_RULE(control getLastRule(), const)
 3 else
 4   ADD_POSTCONDITION_TO_RULE(control.getRuleByIndex(indexOfMarked),
     const)
 5   for i = indexOfMarked + 1 to control.Rules.Length
 6     ADD_PRECONDITION_TO_RULE(control.getRuleByIndex(i), const)
 7     ADD_POSTCONDITION_TO_RULE(control.getRuleByIndex(i), const)
 8   end for
 9 end if
10 return EXECUTE_CONTROL(control, match)
```

The fourth parameter of the guarantee algorithm - $indexOfMarked$ - contains the index of a marked step, this step guarantees the given property, and the steps after this marked step preserve this property. If the given control has exactly one step, or if the given $indexOfMarked$ is less than zero (there is no marked step), the algorithm enlists the given constraint in the postconditions of the last step. If the parameter $indexOfMarked$ contains valid index, the algorithm enlists the given constraint in the postconditions of the step specified by the $indexOfMarked$, and enlists the constraint in both the pre- and the postconditions of the steps $S_{indexOfMarked+1}, \ldots S_n$. Finally the algorithm calls the EXECUTE_CONTROL method and retrieves its return value.

4 Conclusions

Our metamodel-based specification of the rules allows assigning OCL constraints to the rules, and they are able to express local constraints. However, it does not mean that validating them does not involve checking other model elements in the input model. OCL constraints enlisted in the rules have effect on the instances of these rules, on the matched and the replaced subgraphs.

We have shown the relationship between the pre- and postconditions and OCL constraints, and how we can use the OCL constraints enlisted in the rewriting rules to check validation, preservation and guarantee properties, or simply how to check models with the help of metamodel-based graph rewriting. If the developer is familiar with UML, then this method seems quite natural, hence the UML knowledge can be transferred with a little justification.

In this paper the concepts of general validation, general preservation and general guarantee have been discussed. It is presented that if a rewriting rule (a finite sequence of steps) specified adequately with the help of constraints, and the rule (the finite sequence of steps) has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the rewriting rule (the finite sequence of steps) refined with the constraints. It means that the modeler's task is to create adequate rules and specify them fully with constraints (pre- and postconditions), and if the execution of the rewriting rule (finite sequence of steps) finishes successfully, it produces a valid result.

The main limitation of the presented method is the local-nature of the rules. If one wants to specify a constraint for an element, (i) it must be included in a rewriting rule, or (ii) it must be referenced by the OCL traversal expressions assigned to the rule elements. Consequently, this method does not provide an easy way to check global constraints such as dead lock examination. For those parts of the graphs that are not affected by a rewriting rule we can not specify constraints. But there are numerous cases such as the elaborated code generation from statechart model, where the whole right side is generated, thus all the output model elements can be validated.

References

[1] D. Akehurst, O. Patrascoiu, *OCL 2.0 - Implementing the Standard for Multiple Metamodels*, Workshop Proceedings, 6th International Conference on the Unified Modeling Language and its Applications, <<UML>>2003, Electronic Notes in Theoretical Computer Science (ENTCS), October 2003.

[2] D. Blostein, H. Fahmy, A. Grbavec, *Practical Use of Graph Rewriting*, Technical Report No. 95-373, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January, 1995.

[3] Corradini A, Ehrig H, Löwe M, Montanari U, Rossi F, *Abstract Graph Derivations in the DoublePushout Approach.* In H.J. Schneider and H. Ehrig, editors, Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science, vol.776 of Lecture Notes in Computer Science, pp. 86–103. Springer Verlag, 1994.

[4] Ehrig H, *Introduction to the Algebraic Theory of Graph Grammars*, In Graph Grammars and Their Applications to Computer Science and Biology, Springer, Ed. Claus V., Ehrig H., Rozemberg G., Berlin, 1979.

[5] Ehrig H, *Tutorial introduction to the algebraic approach of graph grammars*. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, Proceedings of the 3rd International Workshop on GraphGrammars and Their Application to Computer Science, volume 291 of Lecture Notes in Computer Science, pages 3–14. Springer Verlag, 1987.

[6] Ehrig H, Korff M, Löwe M, *Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts*. In H. Ehrig, H.J. Kreowski, and G. Rozenberg, editors, Proceedings of the 4th International Workshop on GraphGrammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 24–37. Springer Verlag, 1991.

[7] Ehrig H (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol. 2. World Scientific, Singapore, 1997.

[8] Ehrig H, Engels G, Kreowski H-J, Rozemberg (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools*, Vol.2. World Scientific, Singapore, 1999.

[9] Generic Modeling Environment (GME) 2000, http://www.isis.vanderbilt.edu/Projects/gme/default.html

[10] Ali Hamie, John Howse, Stuart Kent, *Interpreting the Object Constraint Language*, Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), December 2-4, 1998, Taipei, Taiwan, 1998.

[11] Karsai G, Agrawal A, Shi F, Sprinkle J, *On the Use of Graph Transformation in the Formal Specification of Model Interpreters*, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.

[12] Levendovszky T, Lengyel L, Charaf H, *Implementing a Metamodel-Based Model Transformation System*, Buletinul Stiintific al Universitatii Politehnica din Timisoara, ROMANIA Seria AUTOMATICA si CALCULATOARE PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE Vol.49 (63), 2004, ISSN 1224-600X.

[13] Levendovszky T, Lengyel L, Mezei G, Charaf H, *A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS*, International Workshop on Graph-Based Tools (GraBaTs) Electronic Notes in Theoretical Computer Science, Rome, 2004.

[14] Levendovszky T, Lengyel L, Charaf H, *Software Composition with a Multipurpose Modeling and Model Transformation Framework*, IASTED 2004, Innsbruck, 2004, pp.590-594.

[15] Sten Loecher, Stefan Ocke, *A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground*, Workshop Proceedings, 6th International Conference on the Unified Modeling Language and

its Applications, <<UML>>2003, Electronic Notes in Theoretical Computer Science (ENTCS) , October 2003.

[16] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, New York, 1988.

[17] MDA Guide Version 1.0.1, OMG, document number: omg/2003-06-01, 12th June 2003 www.omg.org/docs/omg/03-06-01.pdf

[18] Object Constraint Language Specification (OCL), www.omg.org

[19] UML 2.0 Specifications, http://www.omg.org/uml/

[20] PROGRES system can be downloaded from http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html

[21] G. Rozenberg (ed.), *Handbook on Graph Grammars and Computing by Graph-Transformation: Foundations*, Vol.1 World Scientific, Singapore, 1997.

[22] Sprinkle J: *Notes for Model-Integrated Computing*, EECE290O, Berkeley, http://www.eecs.berkeley.edu/ sprinkle/teaching/eecs290o/reading/book/

[23] Sztipanovits J, Karsai G, *Model-Integrated Computing*, IEEE Computer, pp. 110–112, April, 1997.

[24] Sztipanovits J, Karsai G, *Generative Programming for Embedded Systems*, LNCS 2487, pp. 32–49, 2002.

[25] Visual Modeling and Transformation System Web Site http://avalon.aut.bme.hu/ tihamer/research/vmts/