An Approach for Quality Assurance of Model Transformations

Duc-Hanh Dang
Department of Software Engineering,
VNU - University of Engineering and Technology,
144 Xuan Thuy, Hanoi, Vietnam
hanhdd@vnu.edu.vn

Martin Gogolla
Department of Computer Science,
University of Bremen
D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

***Abstract*—Model transformation is an important building block for model-driven approaches. It puts forward a necessity as well as a challenge for validating and verifying transformations. This paper proposes a specification method and an OCL-based framework for model transformations. The approach is based on an integration of Triple Graph Grammars and the Object Constraint Language (OCL) as a formal foundation. The OCL-based transformation framework offers an on-the-fly verification of model transformations and means for transformation quality assurance.**

***Index Terms*—Model Transformation, Graph Transformation, OCL, Validation&Verification, Pre- and Postcondition, Invariant.**

I. Introduction

Model transformation can be seen as the heart of model-driven approaches [1]. Transformations are useful for different goals such as (1) to relate views of the system to each other; (2) to reflect about a model from other domains for an enhancement of model analysis; and (3) to obtain a mapping between models in different languages. In such cases it is necessary to ensure the correctness of transformations. This is also a challenge because of the diversity of models and transformations.

Many approaches to model transformation have been introduced [2]. The works in [3], [4] offer mechanisms for transformations in line with the Query/View/Transformation (QVT) standard [5]. The ideas in [6], [7] focus on the approach based on graph transformation for unidirectional transformations. Triple Graph Grammars (TGGs) [8] are a similar approach for bidirectional transformations. In addition to specification and realization of transformations as proposed by these works, several papers discuss how to ensure the correctness of transformations. In [9] the authors introduce a method to derive Object Constraint Language (OCL) invariants from declarative transformations like TGGs and QVT in order to enable their verification and analysis. The work in [10] aims to establish a framework for transformation testing. Up to now, to offer a suitable approach for model transformation which supports for quality assurance of transformations is still a "hot" issue.

In this paper we focus on an integration of TGGs and OCL as a foundation for model transformation. Incorporating OCL conditions in triple rules allows us to express better transformations. Our approach targets both declarative and operational features of transformations. Within the approach a new method to extract invariants for TGG transformations is introduced. We propose a specification method of transformations and an OCL-based framework for model transformation. We realize the approach in USE [11], a tool with full OCL support. This offers an on-the-fly verification of transformations and means for quality assurance of transformations.

The rest of this paper is structured as follows. Section II explains our basic idea. Section III focuses on the formal foundation for specification and realization of transformations. Section IV explains an OCL-based framework for transformation quality assurance. Section V shows how the approach is realized in the USE tool. Section VI comments on related work. The paper is closed with a conclusion and a discussion of future work.

II. The Basic Idea

We focus on the SC2EHA transformation between a statechart and an extended hierarchical automaton in order to illustrate our approach. Models in this example [12] represent a traffic supervisor system for a crossing of a main road and a country road (Fig. 1). The lamp controller provides higher precedence to the main road as follow: If more than two cars are waiting at the main road (this information is provided by a sensor), the lamp will be switched from red to red-yellow immediately, instead of a waiting period as usual. A camera allows the system to record cars running illegally in the crossing during the red signal.

A. Overview of the Transformation

1) Motivation of the transformation: The UML statechart (state machine) [13] is a light-weight notation for describing the system behavior. In order to model-check statecharts, it is necessary to transform them and represent them in a mathematical formalism like Extended Hierarchical Automata (EHAs). EHAs have been proposed in [14] as an intermediate format to facilitate linking new tools to a statechart-based environment. This formalism uses single-source/single-target transitions (as in usual automata), and forbids interlevel transitions. The EHA notation is a simple formalism with a more restricted syntax than statecharts which nevertheless allows us to capture the richer formalism [14].

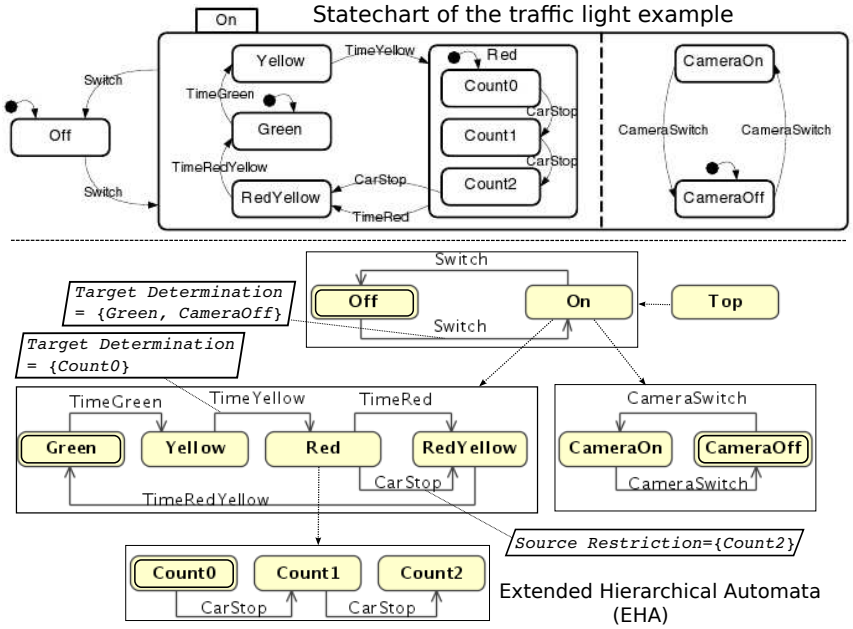


Fig. 1. Statechart and extended hierarchical automaton for the traffic light example (adapted from [12])

2) Example Statechart: Figure 1 shows the example statechart. The system has two basic *states* `On` and `Off`, reflecting whether the system is turned on or off. There is a concurrent decomposition of the `On` state. The region on the left corresponds to the lamp state, and the region on the right corresponds to the camera state. Each of these regions is concurrently active. The `On` state is referred to as an *orthogonal state*, i.e., a *composite state* containing more than one region. The `Off` state, which does not have sub-states, is referred to as a *simple state*.

3) Example Extended Hierarchical Automaton : Figure 1 presents the example EHA as another representation of the example statechart. This EHA consists of four *sequential automata* (denoted by rectangles), each of which contains *simple states* and *transitions*. States can be refined by concurrently operating sequential automata, imposing a tree structure on them. In Fig. 1 the refinement is expressed by the dotted arrows, e.g., the `On` state is refined by two sequential automata.

Interlevel transitions in statecharts are transitions which do not respect the hierarchy of states, i.e., those that may cross borderlines of states. The EHA expresses them using labeled transitions in the automata representing the lowest composite state that contains all the explicit source and target states of the original transition. For example, the interlevel transition from `Count2` to `RedYellow` in the statechart is represented by the transition from `Red` to `RedYellow` together with the label `Count2` in the corresponding automaton. This label is referred as a *source restriction*. The transition is *enabled* only if its source and all state in the source restriction set ({`Count2`}) are active. The interlevel transition from `Off` to `On` is represented by the similar transition in the corresponding automaton together with labels `Green` and `CameraOff`, called a *target determination*. When taking the transition, the target and all states in the target determination set ({`Green`, `CameraOff`}) are *entered* and become active.

4) The SC2EHA Transformation: The transformation needs to show intuitive correspondences between statecharts and EHA models as follows.

- **Initial state.** The initial state of an automaton, e.g., the `Off` state, corresponds to a state of the statechart marked by an initial pseudo state.
- **Simple state.** Each simple state of the statechart corresponds to a unique state of the sequential automaton.
- **Concurrent state.** The concurrent state of the statechart, e.g., the `On` state, is mapped to a state of the sequential automaton. Each region of the concurrent state corresponds to an automaton. These automata refine the state of the sequential automaton.
- **Non-concurrent composite state.** The non-concurrent composite state of the statechart, e.g., the `Red` state, is mapped to the state of the sequential automaton together with an automaton refining it.
- **Transitions.** Each transition in the statechart is uniquely mapped to a transition of the EHA. In case the transition is an interlevel one, the transition of the EHA needs to be labelled with the source restriction and target determination sets.

B. Requirements for Transformation Quality Assurance

Quality assurance for transformations means systematic monitoring and evaluation of various aspects so that the transformation meets modeler expectations. Here we focus on the questions “is the transformation right?” and “is this the right transformation?” that characterize verification and validation.

1) Verification: We need to check if there are any defects in a transformation. Here we consider a transformation as a program taking the source and target model as the input and output respectively. Based on such a transformation model, we could expect several reasonable assumptions to be satisfied. For example, with the SC2EHA transformation, the input statechart and the output EHA model need to fulfill restrictions at the metamodel level for well-formedness criteria. Failing to satisfy such criteria will be a symptom of an incorrect transformation.

2) Validation: The validation of a transformation is the process of applying the transformation in various scenarios and comparing the de facto result with the expected outcome. The process cannot be fully automated: The modeler often has to define relevant scenarios together with the expected outcome, so-called test cases, and then to compare the obtained and expected result. The process often depends on a semi-automated solution, where test cases may be generated automatically, the execution may be animated and debugged, and the difference between the result and the expected outcome may be highlighted.

C. Central Steps of the Methodology

Let us explain the central steps of the transformation development method that we propose: (1) Models as the input and output of transformations are defined by metamodels together with OCL restriction. (2) Our transformation models are established based on the incorporation of triple rules and OCL at two levels, declarative and operational specification of transformations. (3) For a transformation quality assurance framework, the reasonable assumptions of transformations in our focus include the following OCL properties: model properties, invariants of transformations, well-formedness of models, and pre- and postconditions of transformation operations.

III. Specification and Realization of Transformation

This section explains our approach based on TGGs and OCL for specification and realization of transformation.

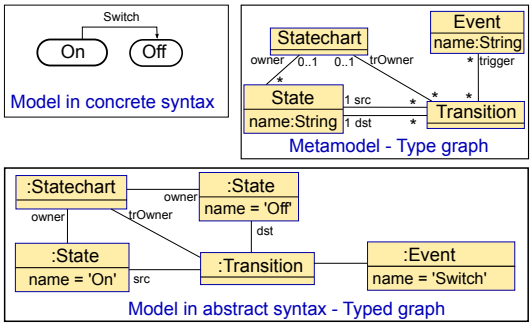


Fig. 2. A typed graph conforms to the metamodel as a type graph

A. Preliminaries of Transformation Based on TGGs

The definitions explained in this section are adapted from the work in [15]. Models in our work are seen as graphs. They are defined by a corresponding metamodel, which is represented as a type graph. Figure 2 shows a simplified metamodel as a type graph which defines the structure of statecharts. The full version of the metamodel is shown in Fig. 7. Instances of the node types (Statechart, State, Transition, and Event) have to be linked according to the edge types between the node types and have to be attributed according to node type attributes.

In order to obtain a mapping between a pair of models, we consider such a combination as a triple graph. Triple graph transformations allow us to build states of the integration.

Definition 1. (Triple Graphs; Triple Graph Morphisms)
Three graphs SG, CG, and TG, called source, connection, and target graph, together with two graph morphisms $s_G : CG \to SG$ and $t_G : CG \to TG$ form a triple graph $G = (SG \xleftarrow{s_G} CG \xrightarrow{t_G} TG)$. G is said to be empty, if SG, CG, and TG are empty graphs. A triple graph morphism $m = (s, c, t) : G \to H$ between two triple graphs $G = (SG \xleftarrow{s_G} CG \xrightarrow{t_G} TG)$ and $H = (SH \xleftarrow{s_H} CH \xrightarrow{t_H} TH)$ consists of three graph morphisms $s : SG \to SH$, $c : CG \to CH$ and $t : TG \to TH$ such that $s \circ s_G = s_H \circ c$ and $t \circ t_G = t_H \circ c$. It is injective, if the morphisms s, c and t are injective. Triple graphs and triple graph morphisms form the category **TripleGraph**.

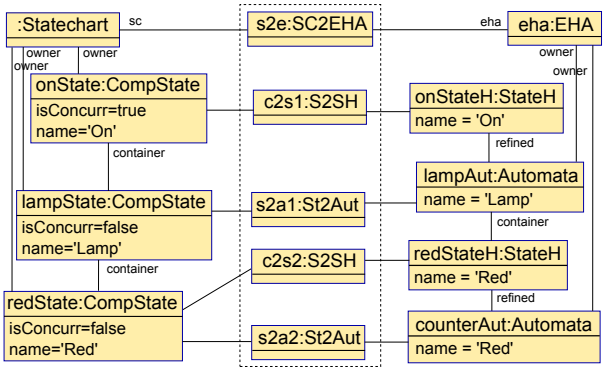


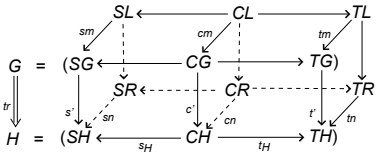
Fig. 3. Triple graph for an integrated SC2EHA model

Example. Triple graph: The graph in Fig. 3 shows a triple graph containing a statechart together with correspondence nodes pointing to the extended hierarchical automata (EHA). References between source and target models denote translation correspondences.

Definition 2. (Triple Graph Grammar)
A triple rule $tr = \mathbf{L} \xrightarrow{tr} \mathbf{R}$ consists of triple graphs $\mathbf{L}$ and $\mathbf{R}$ and an injective triple graph morphisms tr.

$$\begin{array}{ccc} L & = & (SL \xleftarrow{s_L} CL \xrightarrow{t_L} TL) \\ {\scriptstyle tr}\downarrow & & {\scriptstyle s}\downarrow \quad {\scriptstyle c}\downarrow \quad {\scriptstyle t}\downarrow \\ R & = & (SR \xleftarrow{s_R} CR \xrightarrow{t_R} TR) \end{array}$$

Given a triple rule $tr = (s, c, t) : \mathbf{L} \rightarrow \mathbf{R}$, a triple graph G and a triple graph morphism $m = (sm, cm, tm) : \mathbf{L} \rightarrow G$, called triple match m, a triple graph transformation step $G \overset{tr,m}{\Longrightarrow} H$ from G to a triple graph H is given by three objects SH, CH and TH in category **Graph** with induced morphisms $s_H : CH \rightarrow SH$ and $t_H : CH \rightarrow TH$. Morphism $n = (sn, cn, tn)$ is called comatch.



A triple graph grammar is a structure $TGG = (TG, S, TR)$ where TG is a triple type graph, S is an initial graph, and $TR = \{tr_1, tr_2, ..., tr_n\}$ is a set of triple rules. Triple graph language of TGG is the set $\{G | \exists$ *triple graph transformation* $S \Rightarrow^* G\}$.

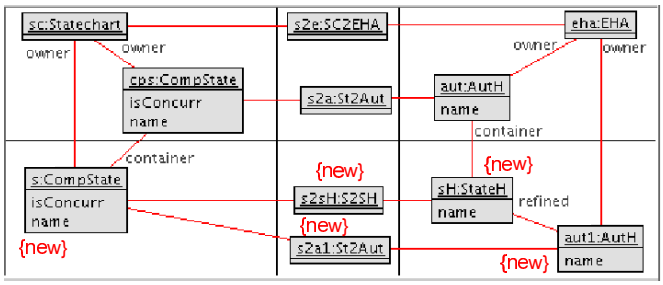


Fig. 4. Triple rule for the SC2EHA transformation

Example. Triple rule: The rule in Fig. 4 is part of a triple graph grammar that generates statecharts and corresponding EHA models. This rule may create a simple state of a statechart and its corresponding state of the corresponding EHA model at any time.

A triple rule allows us to derive new rules for forward and backward transformation, model integration, and model co-evolution. Let $TGG = (TG, S, TR)$ be a triple graph grammar, VL be the language of TGG, and VL_s, VL_c, and VL_t be the source, correspondence, and target language as the result of the projection onto the source, correspondence, and target part of VL, respectively.

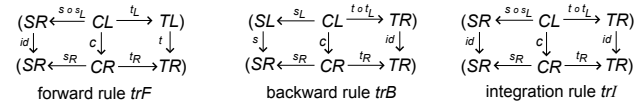
Definition 3. (Forward Transformation)
Let a graph $G_S \in VL_s$ be given. A forward transformation from G_S to G_T is a computation to define the graph $G_T \in VL_t$ through a triple derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.

Definition 4. (Backward Transformation)
Let a graph $G_T \in VL_t$ be given. A backward transformation from G_T to G_S is a computation to define the graph $G_S \in VL_s$ through a derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.

Definition 5. (Model Integration)
Let the graphs $G_S \in VL_t$ and $G_T \in VL_s$ be given. A model integration of G_S and G_T is a computation to define a derivation $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.

Definition 6. (Model Co-Evolution)
Let $E_S \in VL_s$ and $E_T \in VL_t$ be graphs as source and target parts of a triple graph E, respectively. A model co-evolution from (E_S,E_T) to (F_S,F_T) is a computation to define graphs $F_S \in VL_s$ and $F_T \in VL_t$ through the derivation $(E_S \leftarrow E_C \rightarrow E_T) \overset{*}{\Rightarrow} (F_S \leftarrow F_C \rightarrow F_T)$.

Definition 7. (Derived Triple Rules)
Each triple rule $tr = L \rightarrow R$ derives forward, backward, and integration rules as follows:



where id is the identify function.

Theorem 1. (Derived Rules for Forward Transformation) *Let $TGG = (TG, S, TR)$ be a triple graph grammar and $(G_S \leftarrow S_C \rightarrow S_T)$ be a triple graph typed by TG. We can define a forward transformation from G_S to G_T as the following conditions are fulfilled.*

(i) $(G_S \leftarrow S_C \rightarrow S_T) \overset{trF_1,m_1}{\Longrightarrow} \ldots \overset{trF_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$, *where $m_i = (sm_i, cm_i, tm_i)$ are triple matches.*

(ii) $\forall i > 0, 0 < j < i$, $sn_j(SR_{tr_j} \setminus SL_{tr_j}) \cap sn_i(SR_{tr_i} \setminus SL_{tr_i}) = \emptyset$, *where (sn_i, cn_i, tn_i) is the comatch of m_i.*

Proof: Suppose that at the i^{th} step of the transformation in (ii), we can define the triple graph G^i such that $S = (S_S \leftarrow S_C \rightarrow S_T) \overset{tr_1,m_1}{\Longrightarrow} \ldots \overset{tr_i,m_i}{\Longrightarrow} G^i = (G^i_S \leftarrow G^i_C \rightarrow G^i_T)$ and $G^1 \subset G^2 \ldots \subset G^i \subset G_S$. Now at the $i + 1^{th}$ step the condition (ii) allows us to define G^{i+1} such that $G^i \subset G^{i+1} \subset G_S$ and $G^i = (G^i_S \leftarrow G^i_C \rightarrow G^i_T) \overset{tr_{i+1},m_{i+1}}{\Longrightarrow} G^{i+1} = (G^{i+1}_S \leftarrow G^{i+1}_C \rightarrow G^{i+1}_T)$. Therefore, by indution there exists a transformation $S = (S_S \leftarrow S_C \rightarrow S_T) \overset{tr_1,m_1}{\Longrightarrow} \ldots \overset{tr_n,m_n}{\Longrightarrow} G^n = (G_S \leftarrow G_C \rightarrow G_T)$. This is what we need to prove. ■

For the backward, and integration transformation between G_S and G_T, we can obtain a similar result. The (ii) condition in this case is shown respectively as follow.
$(S_S \leftarrow S_C \rightarrow G_T) \overset{trB_1,m_1}{\Longrightarrow} \ldots \overset{trB_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$,
$(G_S \leftarrow S_C \rightarrow G_T) \overset{trI_1,m_1}{\Longrightarrow} \ldots \overset{trI_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$

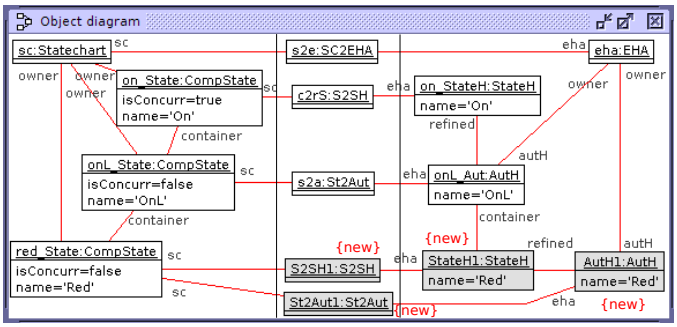


Fig. 5. A forward transformation step by the forward rule derived from the rule shown in Fig. 4

Example. Figure 5 shows a transformation step for the forward

transformation from a statechart to an EHA model. The forward rule is derived from the rule shown in Fig. 4

B. Incorporation of OCL and Triple Rules

We propose to employ OCL conditions as restrictions on the applicability of triple rules. It allows us to increase the expressiveness of triple rules. For example, with the rule shown in Fig. 3, we could attach it with the OCL precondition $cps.isConcurr = false$ and the postcondition $s.name <> oclUndefined(String) \wedge aut1.name <> oclUndefined(String)$. OCL application conditions of a triple rule can be defined as a combination of OCL conditions in parts of the triple rule. Formally, they are defined as follows.

Definition 8. (OCL Application Conditions)
OCL application conditions (BACs[1]) of a triple rule consist of OCL conditions in source, target, and correspondence parts of the triple rule. BACs within the LHS and RHS of the triple rule are application pre- and postconditions, respectively:

- $BAC_{pre} = BAC_{SL} \cup BAC_{CL} \cup BAC_{TL}$,
- $BAC_{post} = BAC_{SR} \cup BAC_{CR} \cup BAC_{TR}$, and
- $BAC = [BAC_{pre}, BAC_{post}]$,

where BAC_{xx} with $xx \in \{`SL', `SR', `CL', `CR', `TL', `TR'\}$ are BACs in the LHS and RHS of the source, correspondence, and target parts of the triple rule, respectively; BAC_{pre} and BAC_{post} are application pre- and postconditions, respectively.

Definition 9. (Application Condition Fulfillment)
A triple rule with BACs is a tuple $tr = (L, R, BAC)$, where BAC includes OCL application conditions. A triple graph H is derived from a triple graph G by a triple rule $tr = (L, R, BAC)$ and a triple match m iff:

- H is derived by $(L \rightarrow R, m)$ and
- BAC is fulfilled in the rule application $G \overset{r}{\Rightarrow} H$,

where $r : L \rightarrow R$ is the rule which is obtained by viewing the triple graphs LHS and RHS of the tr rule as plain graphs.

Theorem 2. (Pre- and Postconditions of Derived Rules)
Let a triple rule tr be given. Preconditions of triple rules derived from tr are defined as follows.

- $BAC^{trF}_{pre} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TL}$,
- $BAC^{trB}_{pre} = BAC^{tr}_{SL} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TR*}$, *and*
- $BAC^{trI}_{pre} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TR*}$

Postconditions of derived rules are defined as follows.

- $BAC^{trF}_{post} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR}$,
- $BAC^{trB}_{post} = BAC^{tr}_{SR} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR*}$, *and*
- $BAC^{trI}_{post} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CR} \cup BAC^{tr}_{TR*}$,

where

- BAC^{trF}_{pre}, BAC^{trB}_{pre}, *and* BAC^{trI}_{pre} *are the precondition of derived rules for forward, backward, and integration transformation, respectively;* BAC^{trF}_{post}, BAC^{trB}_{post}, *and* BAC^{trI}_{post} *are the postcondition of the derived rules,*
- BAC^{tr}_{xx} *with* $xx \in \{`SL', `SR', `CL', `CR', `TL', `TR'\}$ *are BACs in the LHS and RHS of parts of the triple rule tr, respectively, and*
- BAC^{tr}_{SR*}, BAC^{tr}_{CR*}, *and* BAC^{tr}_{TR*} *are BACs excepting ones with '@pre' in SR, CR, and TR , respectively.*

Proof: According to Theorem 1, a derivation with derived triple rules, e.g., $(M_S \leftarrow \phi \rightarrow \phi) \overset{trF_1,m_1}{\Longrightarrow} \ldots \overset{trF_n,m_n}{\Longrightarrow} (M_S \leftarrow M_C \rightarrow M_T)$, is always defined in relation to a corresponding derivation with the original triple rules, i.e., $(\phi \leftarrow \phi \rightarrow \phi) \overset{tr_1,m_1}{\Longrightarrow} \ldots \overset{tr_n,m_n}{\Longrightarrow} (M_S \leftarrow M_C \rightarrow M_T)$. Therefore, we can formulate pre- and postconditions of derived triple rules in such a way. ■

C. The RTL Transformation Language

We define the RTL[2] language in order to specify triple rules incorporating OCL. The declarative specification in textual form can generate the different operations for transformation scenarios as explained in Subsect. III-A. We realize the operations by taking two views on them: Declarative OCL pre- and postconditions are employed as operation contracts, and imperative command sequences are taken as an operational realization. Figure 6 illustrates the RTL specification of triple rules and the generated corresponding OCL operations.

IV. Quality Assurance of Transformations

This section discusses how our OCL-based transformation framework offers means for transformation quality assurance.

A. Verification of Transformation

We explain the aim in a formal way: Let MM_S and MM_T be metamodels for source and target models, respectively. Let $TGTS = (S, TR)$ be a triple graph transformation system, which relates source and target models to each other. With respect to $TGTS$ we define an RTL specification of the transformation. For each source model M_S the RTL specification allows us to define a corresponding target model M_T by forward operations. Note that we only focus on forward transformation since the verification for other transformation scenarios can be similarly obtained. We need to check if the target model M_T is correctly defined.

1) Check invariants of transformations: We can see triple rules as templates establishing mappings between source and target models. Therefore, the transformation is correct only if such mappings conform to triple rules. For example, with the triple rule shown in Fig. 4 a mapping that conforms to the rule must include 11 objects and 14 links. For the check we aim to maintain "traces" for such mappings. We propose to add a new node into the correspondence part of each rule. The new node represents an instance of a class whose name coincides with the rule name. The node is linked to all nodes in the correspondence part so that from this node we can navigate to them within an OCL expression. We can define an OCL condition in order to represent the pattern of this rule. For example, the following OCL condition represents for the triple rule $CompStateNest$ shown in Fig. 4: An invariant of the $CompStateNest$ class is defined. The transformation is correct only if such an invariant are valid.

[1]BACs stands for Boolean Application Conditions

[2]RTL stands for Restricted Graph Transformation Language

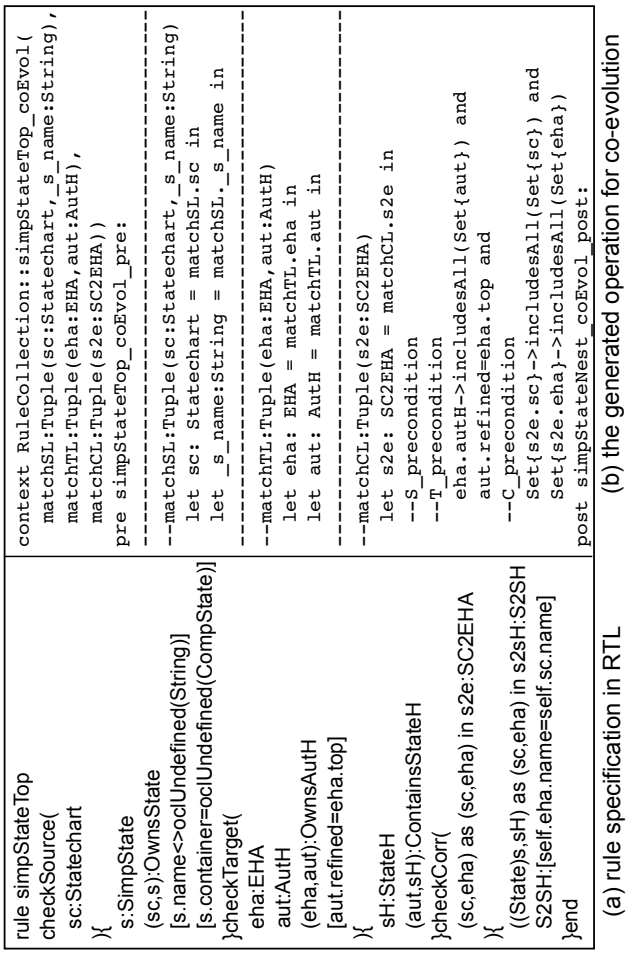


Fig. 6. RTL specification of triple rules and generated OCL operations

```
context CompStateNest inv isMatch:
let s2e:SC2EHA = self.s2e in
let s2a:St2Aut = self.s2a in
let s2sH:S2SH = self.s2sH in
let s2a1:St2Aut = self.s2a1 in
s2e.isDefined and s2a.isDefined and
s2sH.isDefined and s2a1.isDefined and
s2e.includes(sc) and s2e.includes(eha) and
s2a.includes(cps) and s2a.includes(autH) and
s2sH.includes(s) and s2sH.includes(sH) and
s2a1.includes(s) and s2a1.includes(aut1) and
s2a.aut.includes(sH) and s2a.aut.includes(eha)
and s2a1.aut1.includes(sH) and
s2a1.aut1.includes(eha) and s2a.cps.includes(s)
and s2e.sc.includes(s) and s2e.sc.includes(cps)
```

2) Check contract fulfillment of transformation steps: According to the algorithm for translating triple rules into OCL operations [16], it follows that the sequence of operation applications for a transformation corresponds to a triple derivation for forward transformation: $d_{tr} : (M_S \leftarrow \phi \rightarrow \phi) \overset{trF_1,m_1}{\Longrightarrow} \ldots \overset{trF_n,m_n}{\Longrightarrow} (M_S \leftarrow M_C \rightarrow M_T)$, where trF_i are forward rules and m_i are triple matches. In order to check the correctness of the transformation we check if each operation application realizes correctly a rule application. By checking the contract of the operation, i.e., a pair of pre- and postconditions it allows us to ensure the correctness of the transformation step. It offers an on-the-fly verification for different transformation properties.

3) Check model properties: The declarative language OCL allows us to navigate and to evaluate queries on models. Therefore, we can employ OCL to express properties of models at any specific moment in time. For example, the following OCL condition expresses the property "*There is a transition from the 'Red' state to the 'Yellow' state.*"

```
Trans.allInstances()->exists(t|
  t.src.name='Red' and
  t.dst.name='Yellow')
```

4) Check well-formedness of models: The transformation with triple rules may maintain the conformance relationship between a model as a typed graph and its metamodel as a type graph. However, when the metamodel is restricted by OCL conditions, models during a transformation may no longer conform to their metamodel. A model conforms to the metamodel, i.e., it is well-formed only if such restricting invariants are fulfilled. For example, during the SC2EHA transformation, the following invariant `ownsChildState` needs to be valid. The invariant expresses the condition "*Every child state of a composite state belongs to the same statechart with the parent state.*"

```
context Statechart inv ownsChildState:
self.state->forAll(p:State|
 if p.oclIsTypeOf(CompState) then
  p.oclAsType(CompState).content->
  forAll(c:State|self.state->includes(c))
 else true endif)
```

B. Validation of Transformation

This section focuses on features of the RTL transformation that might provide support for a semi-automated solution to validate transformations.

1) Model integration for test cases: Given a test case including the source model M_S and the expected target model M_T. To check the transformation with the test case means we check if M_T coincides with the resulting model $M_T^{'}$. Instead of this, we could employ integration rules in order to obtain an integration of M_S and M_T: A mapping between these models is established. The derivation is such that $(M_S \leftarrow \phi \rightarrow M_T) \overset{trI_1,m_1}{\Longrightarrow} \ldots \overset{trI_n,m_n}{\Longrightarrow} (M_S \leftarrow M_C \rightarrow M_T)$, where trI_i are integration rules and m_i are triple matches. In this way the transformation can be better animated for the modeler.

2) Animation of transformation: After each transformation step, we can see the combination of the source, correspondence, and target part as a whole model. We could employ OCL expressions in order to explore such a model. Mappings within the current rule application can be highlighted by OCL queries. This makes it easier for the modeler to check if the rule application is correct.

V. Tool Support

Our approach for verification and validation of transformation is realized with the support of USE [11], which is a tool for analysis, reasoning, verification and validation of UML/OCL specifications. Specifically, USE allow us to check

class invariants, pre- and postconditions of operations, and properties of models, which are expressed in OCL. In USE system states are represented as object diagrams. System evolution can be carried out using operations based on basic state manipulations, such as (1) creating and destroying objects or links and (2) modifying attributes. In this way a framework for model transformation based on the integration of TGGs and OCL are completely covered by USE. Figure 7 shows metamodels for the SC2EHA transformation in USE. Due to space limitations, the full realization for the transformationis only shown in the long version of this paper [17].

The RTL specification of a transformation is translated into transformation operations in OCL. The operation is realized by taking two views on it: Declarative OCL pre- and postconditions are employed as operation contracts, and USE command sequence are taken as an operational realization. With the full OCL support, USE allows us to realize the verification and validation of transformations as discussed in Sect. IV.

VI. Related Work

Triple Graph Grammars (TGGs) have been proposed in [8]. Since then, many works have extended TGGs for software engineering [18]. Here we focus on the incorporation of TGGs and OCL as a foundation for transformations as proposed in our previous work [19], [16]. Note that our previous work mostly focuses on how triple rules incorporating OCL are operationalized. In this paper, we concentrate on the correctness and validation of such a transformation, towards an OCL-based framework for transformation quality assurance. Contributions in this paper include the following points: (1) to define pre- and postconditions of triple rules, which are derived from TGG rules incorporating OCL (specified in the RTL language), (2) an enhancement to extract invariants from declarative TGG rules so that we could check if a transformation step is valid, and (3) a discussion of a framework for quality assurance of transformation.

Many approaches have been proposed for model transformation. Most of them are in line with the standard QVT [5] such as ATL [3] and Kermeta [4]. Like our work, they allow the developer to precisely present models using metamodels and OCL. The advantage of our approach is that it is based on the integration of TGGs and OCL, which allows the developer to automatically analyze and verify transformations, and supports for bidirectional model transformation.

Our approach for model transformation is based on graph transformation like the work in VMTS [6] and Fujaba [18]. Many other works focus on the translation of the transformation to a formal domain for model checking such as Alloy in [20], Promela in [21], and Maude in [22].

In [9] the authors propose a method to derive OCL invariants from TGG and QVT transformations in order to enable their verification and analysis. Our approach targets to support for both declarative and operational features of transformations. We also introduce a new method to extract invariants for TGG transformations. Several other works focus on approaches for verification and validation of transformation. The proposal

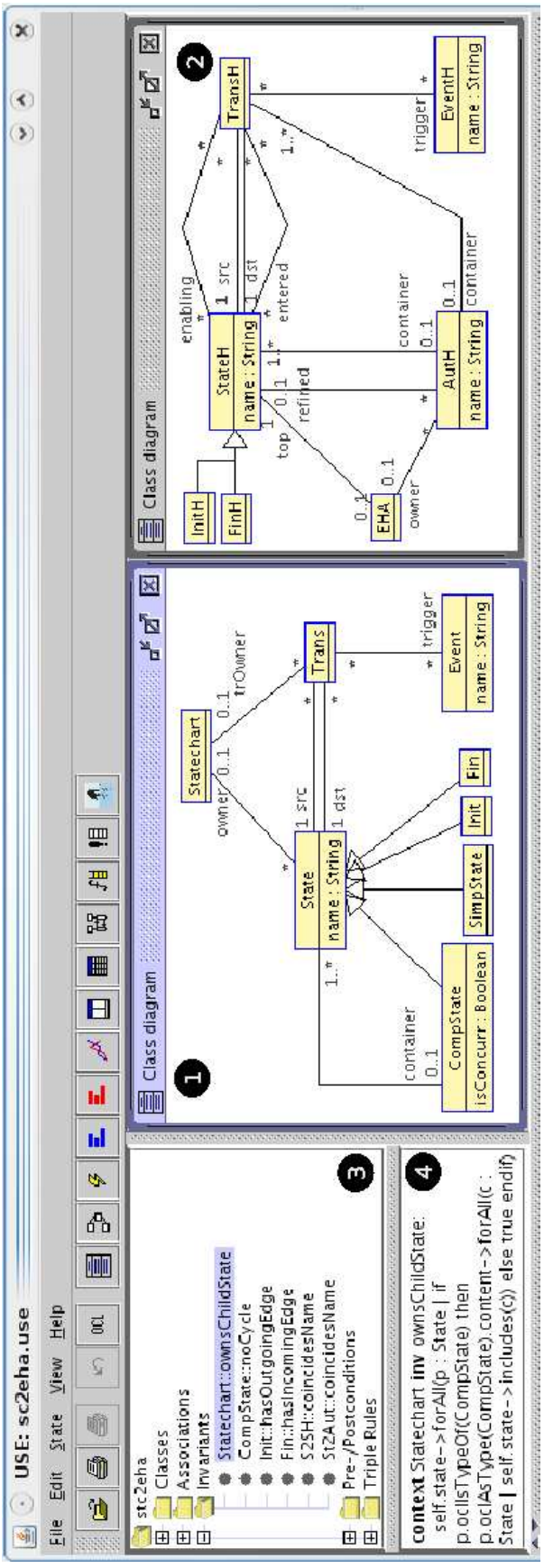


Fig. 7. Metamodels for the SC2EHA transformation

in [23] introduces a method to check semantic equivalence between the initial model and the generated code. The approach in [7] verifies transformation correctness with respect to semantic properties by model checking the transition system of the source and target models. The work in [10] aims at developing frameworks for transformation testing.

VII. Conclusion

We have introduced an approach for quality assurance of model transformations: (1) The foundation of the approach is based on the integration of TGGs and OCL. We have further formulated operation contracts for derived triple rules in order to realize them as OCL operations with the two views: Declarative OCL pre- and postconditions are employed as operation contracts, and imperative command sequences are taken as an operational realization. (2) Both declarative and operational views are obtained by an automatic translation from the specification of transformations into the RTL language. (3) This work also embodies a new method to extract invariants for transformations. The central idea is to view transformations as models. (4) An OCL-based framework for model transformation is established. As being realized on a full OCL support environment like USE, the framework offers a support for validation and verification of transformations.

Our future work includes the following issues. We aim to enhance the technique to extract invariants for transformation models. A control structure like sequence diagram for the RTL specification is also in the focus of our future work. The goal is to increase the efficiency of transformations. The technique to generate test cases from the RTL specification will also be explored. We will focus on other properties of transformations such as the determinateness of transformation. These are efforts towards a full framework for quality assurance of model transformations. Larger case studies must give detailed feedback on the proposal.

Acknowledgement

This work has been supported by the project CN.11.03 of VNU-University of Engineering and Technology. We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

[1] S. Sendall and W. Kozaczynski, "Model Transformation: the Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, pp. 42– 45, 2003.

[2] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[3] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, Jun. 2008.

[4] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-Oriented Meta-languages," in *Model Driven Engineering Languages and Systems*, vol. 3713. Springer Berlin, 2005, pp. 264–278.

[5] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/07-07-07*. OMG, 2007.

[6] L. Lengyel, T. Levendovszky, and H. Charaf, "Validated Model Transformation-Driven Software Development," *Int. J. Comput. Appl. Technol.*, vol. 31, no. 1/2, pp. 106–119, 2008.

[7] D. Varró and A. Pataricza, "Automated Formal Verification of Model Transformations," in *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, Eds. Technische Universität München, 2003, pp. 63–78.

[8] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, ser. LNCS, M. Schmidt, Ed., vol. 903. Springer-Verlag, 1995, pp. 151–163.

[9] J. Cabot, R. Clarisó, E. Guerra, and J. d. Lara, "Verification and Validation of Declarative Model-to-model Transformations Through Invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[10] Y. Lin, J. Zhang, and J. Gray, "A Framework for Testing Model Transformations," in *Model-driven Software Development - Research and Practice in Software Engineering*, S. Beydeda, M. Book, and V. Gruhn, Eds. Springer, 2005, pp. 219–236.

[11] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-Based Specification Environment for Validating UML and OCL," *Science of Computer Programming*, 2007.

[12] G. Pintér and I. Majzik, "Modeling and Analysis of Exception Handling by Using UML Statecharts," in *Scientific Engineering of Distributed Java Applications*, vol. 3409. Springer Berlin, 2005, pp. 58–67.

[13] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual, 2nd Edition*. Addison-Wesley Professional, 2004.

[14] E. Mikk, Y. Lakhnechi, and M. Siegel, "Hierarchical automata as model for statecharts," in *Advances in Computing Science*, vol. 1345. Springer Berlin, 1997, pp. 181–196.

[15] H. Ehrig, C. Ermel, and F. Hermann, "On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars," in *Proceedings of the Third International Workshop on Graph and Model Transformations*. Leipzig, Germany: ACM, 2008, pp. 9–16.

[16] D.-H. Dang and M. Gogolla, "On Integrating OCL and Triple Graph Grammars," in *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, M. Chaudron, Ed., vol. 5421. Springer, 2009, pp. 124–137.

[17] D. Dang and M. Gogolla, *An Approach for Quality Assurance of Model Transformations (Long Version)*. VNU Report, http://www.uet.vnu.edu.vn/ hanhdd/publications/rtl.pdf, 2012.

[18] J. Greenyer and E. Kindler, "Reconciling TGGs with QVT," in *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, ser. LNCS, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 16–30.

[19] D.-H. Dang and M. Gogolla, "Precise Model-Driven Transformation Based on Graphs and Metamodels," in *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November, 2009*, D. V. Hung and P. Krishnan, Eds. IEEE Computer Society Press, 2009, pp. 307–316.

[20] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of Model Transformations via Alloy," in *Proceedings of 4th Workshop on Model-Driven Engineering, Verification and and Validation (MoDeVVA'07)*, 2007, pp. 47–56.

[21] D. Varró, "Automated Formal Verification of Visual Modeling Languages by Model Checking," *Software and Systems Modeling*, vol. 3, no. 2, pp. 85–113, 2004.

[22] J. E. Rivera, E. Guerra, J. d. Lara, and A. Vallecillo, "Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude," in *Software Language Engineering*, vol. 5452. LNCS, 2008, pp. 54–73.

[23] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner, "Towards Verified Model Transformations," in *Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV2a), Genova, Italy*, D. Hearnden, J. G. Süß, B. Baudry, and N. Rapin, Eds. Le Commissariat l'Energie Atomique - CEA, 2006, pp. 78–93.