VNU Journal of Science: Comp. Science & Com. Eng., Vol. 32, No. 1 (2016) 42–57

An OCL-Based Framework for Model Transformations

Duc-Hanh Dang[1,*], Martin Gogolla[2]

[1]*VNU University of Engineering and Technology, Hanoi, Vietnam*
[2]*University of Bremen, Bremen, Germany*

Abstract

Model transformation is an important building block for model-driven approaches. It puts forward a necessity and a challenge to specify and realize model transformation as well as to ensure the correctness of transformations. This paper proposes an OCL-based framework for model transformations. The formal foundation of the framework is the integration of Triple Graph Grammars and the Object Constraint Language (OCL). The OCL-based transformation framework offers an on-the-fly verification of model transformations and means for transformation quality assurance.

Received 06 December 2015, revised 25 December 2015, accepted 31 December 2015

Keywords: Model Transformation, OCL, Validation & Verification, Precondition and Postcondition, Invariant.

1. 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. Within such cases it is necessary to offer methods to specify and realize model transformation as well as to ensure the correctness of transformations. This is really a challenge because of the diversity of models and transformations.

Problem. Many approaches to model transformation have been introduced, as surveyed in [2]. The works in [3, 4] offer mechanisms for model transformations in line with the Query/View/Transformation (QVT) standard [5]. The ideas in [6, 7] focus on the graph transformation-based approach for unidirectional transformations. Triple Graph Grammars (TGGs) are proposed in [8] as 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. To the best of our knowledge, so far there has not been any suitable approach yet to support both specification and quality assurance of transformations.

Contribution. (1) In this paper we focus on the integration of TGGs and OCL as a foundation for model transformation. Incorporating OCL conditions in triple rules of a triple graph grammar allows us to express better transformations. (2) Our approach targets both declarative and operational features of transformations. Within the approach a new method to extract invariants for TGG

* Corresponding author. Email: hanhdd@vnu.edu.vn

transformations is introduced. (3) 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.

Outline. The rest of this paper is structured as follows. Section 2 motivates our work by means of examples. Section 3 introduces a formal foundation for specification and realization of transformations. Section 4 proposes an OCL-based framework for model transformations. Section 5 explains how the framework could offer means for transformation quality assurance. Section 6 shows how the approach is realized in the USE tool. Section 7 comments on related work. The paper is closed with a conclusion and a discussion of future work.

2. Motivating Example

We focus on a transformation situation between a statechart and an extended hierarchical automaton. A UML statechart (state machine) [12] offers 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 [13] 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 [13]. Figure 1 presents models for a traffic supervisor system for a crossing of a main road and a country road [14]. 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 obeying a waiting period as usual. A camera allows the system to record cars running illegally in the crossing during the red signal period.

The example statechart (Fig. 1) shows 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. Note that in our specification the synchronization between the two regions currently currently is not reflected. The `Off` state, which does not have sub-states, is referred to as a *simple state*.

The example EHA is 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. 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`. 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.

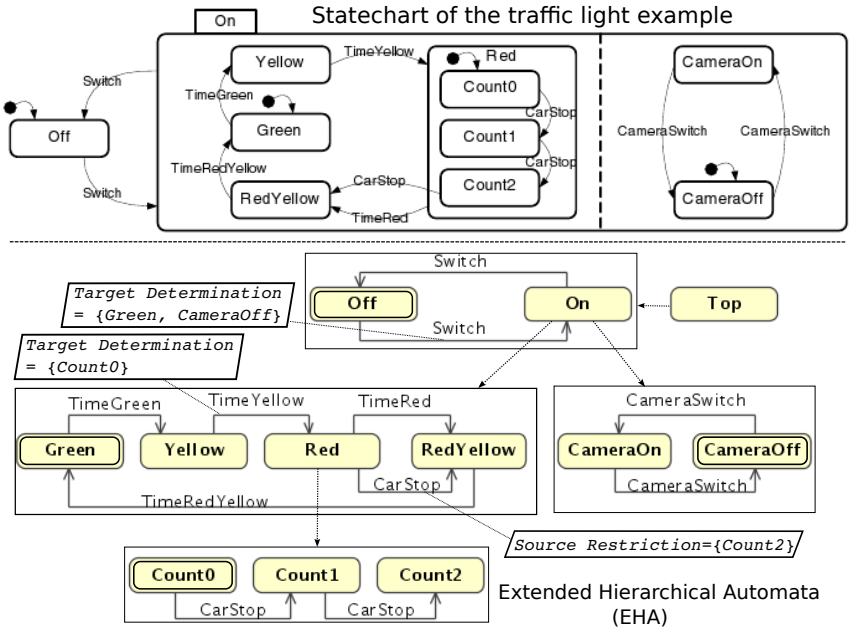


Fig. 1: Statechart and extended hierarchical automaton for the traffic light example. (adapted from [14])

The transformation example motivates our work with the following requirements: (1) Specifying and realizing of transformations: We need to offer means so that transformations could be specified and performed, as well as source and target models could be represented and taken as the input and output of transformations. (2) Verifying transformations: We need to check if there are any defects in a transformation. A transformation can be considered as a program taking the source and target model as the input and output. We could expect several reasonable assumptions on such a transformation model to be satisfied. (3) Validating transformations: The aim is to ensure a transformation is a "right" one by executing the transformation in various scenarios and comparing the de facto result with the expected outcome. The process cannot be fully automated: The modeler has to define relevant scenarios, so-called test cases, and then to compare the obtained and expected result.

3. Foundations for a Model-Driven Approach

This section explains foundations for a model-driven approach to software engineering.

3.1. Graph Transformation

Definition 1 (Graphs and Morphisms). *Let a set of labels L be given.* A directed, labeled graph *is a tuple* $G = (V_G, E_G, s_G, t_G, lv_G, le_G)$*, where*

- V_G *is a finite set of nodes (vertices),*
- $E_G \subseteq V_G \times V_G$ *is a binary relation describing the edges,*
- $s_G, t_G : E_G \rightarrow V_G$ *are source and target functions mapping graph edges to nodes, and*
- $lv_G : V_G \rightarrow L$ *and* $le_G : E_G \rightarrow L$ *are functions that assign a label to a node and an edge, respectively.*

A graph is said to be empty, if the V_G *and* E_G *are empty sets.*

Let two directed, labeled graphs $G = (V_G, E_G, s_G, t_G, lv_G, le_G)$ *and* $H = (V_H, E_H, s_H, t_H, lv_H, le_H)$ *be given.* A graph

morphism $f : G \to H$ *is a pair* (f_V, f_E)*, where* $f_V : V_G \to V_H$*,* $f_E : E_G \to E_H$ *preserves sources, targets, and labels, i.e.,* $f_V \circ s_G = s_H \circ f_E$ *and* $f_V \circ t_G = t_H \circ f_E$*.*

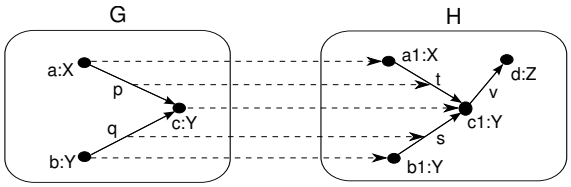


Fig. 2: Two directed, labeled graphs G and H and a graph morphism $G \to H$ (informally speaking, H contains G).

Figure 2 shows two directed, labeled graphs G and H and a graph morphism $G \to H$. The mapping is represented by dashed lines between the nodes and edges of G and H.

Triple graph grammars (TGGs) have been proposed as a means to specify bidirectional translations between graph languages. Integrated graphs obtained by triple derivations are called triple graphs.

Definition 2 (Triple Graphs and 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* $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$*. 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* $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$ *and* $H = (SH \overset{s_H}{\leftarrow} CH \overset{t_H}{\rightarrow} TH)$ *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.*

Figure 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.

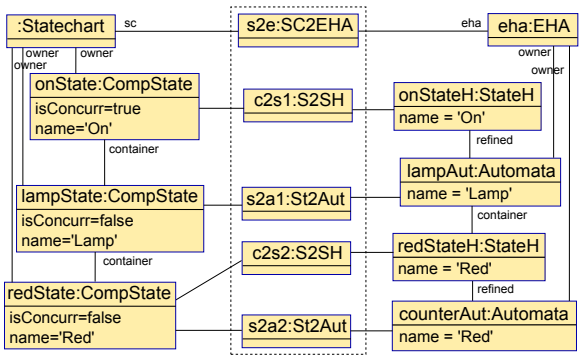


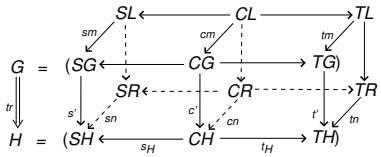
Fig. 3: Triple graph for an integrated SC2EHA model.

Definition 3 (Triple Graph Grammar). *A triple rule* $tr = \mathbf{L} \overset{tr}{\rightarrow} \mathbf{R}$ *consists of triple graphs* $\mathbf{L}$ *and* $\mathbf{R}$ *and an injective triple morphisms tr.*

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

An application of a triple rule $tr = (s, c, t) : L \to R$ *to a given triple graph G to yield a triple graph H consists of the following steps:*

- *Choose an occurrence of the LHS L in G by defining a triple graph morphism* $m = (sm, cm, tm) : L \to G$*, called a* triple match*.*
- *Glue the graph G and the RHS R according to the occurrences of L in G so that new items that are presented in R but not in L are added to G. This yields a gluing graph Z. Formally, we have* $m(L \cap R) = G \cap Z$*. Here, graphs can be seen as a set of items including vertices and edges.*
- *Remove the occurrence of L from Z as well as all* dangling edges*, i.e., all edges incident to a removed node. This yields the resulting graph H. Formally, we have* $H = Z \setminus m(L)$*.*
- *The gluing step allows us to obtain the so-called comatch morphism* $n = (sn, cn, tn)$*, where* $sn = SR \to SH$*,* $cn = CR \to CH$*, and* $tn = TR \to TH$*. The induced morphisms* $s_H : CH \to SH$ *and* $t_H : CH \to TH$ *could be obtained from the comath morphism n.*



A triple graph grammar is a structure TGG = (TG, S, TR) where TG includes a so-called triple type graph and a typing function mapping so-called typed graphs to the type graph, S is an initial graph, and TR = {tr₁, tr₂, ..., trₙ} is a set of triple rules. Triple graph language of TGG is the set {G | G is typed by TG ∧ ∃ triple graph transformation S ⇒* G}.*

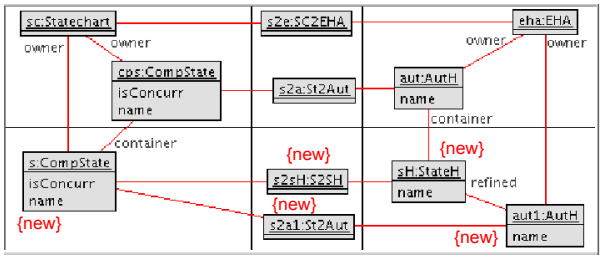
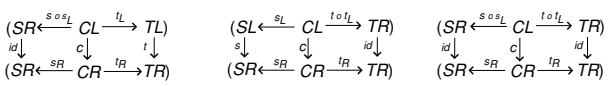


Fig. 4: Triple rule for the SC2EHA transformation.

Figure 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.

Such an integrated triple graph is often defined by a triple derivation. Derived triple rules allows us to compute the triple graph by taking the source (target or both) model as the input. A detailed explanation of how to apply derived triples is shown in SubSect. 3.5.

Definition 4 (Derived Triple Rules). *Each triple rule tr = L → R derives forward, backward, and integration rules as follows:*



where id is the identify function.

3.2. The Object Constraint Language

The Object Constraint Langauge (OCL) [15] is a formal language to describe expressions on UML models, e.g., as shown in Fig. 5: (1) OCL expressions, that might be object constraints or queries, do not have side effects. (2) The OCL is a typed language. Each valid (well-formed) OCL expression has a type, which is the type of the evaluated value of this expression. The type system of OCL includes basic types (e.g., `Integer`, `Real`, `String`, and `Boolean`), object types, collection types (e.g., `Collection(t)`, `Set(t)`, `Bag(t)`, and `Sequence(t)` for describing collections of values of type `t`), and message types. (3) OCL is often employed for the following purposes:

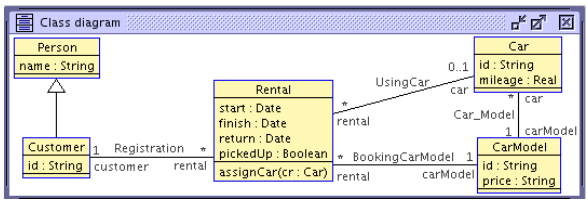


Fig. 5: Object model visualized by a class diagram.

- To specify invariants, i.e., conditions that must be true for all instances of the class in all system states. Example:

```
-- The number of cars is
-- greater than 10.
context CarModel inv:
self.car.size() > 10
```

- To describe pre- and post conditions on operations.

```
-- When a car is picked up.
context Rental::assignCar(cr:Car)
pre self.car = null
post self.car = cr
```

- To describe guards within a statechart.

- As a query language, i.e., to query the given system state by OCL expressions.

OCL expressions are developed on the basis of an object model. The aim is to allow us to express attribute values and logic conditions on the structure defined by the object model. Specifically, the object model structure is extended with an OCL algebra [16].

Definition 5 (Object Models).
An object model is the structure
$\mathcal{M}$ = (CLASS, ATT_c, OP_c, ASSOC, associates, roles, multiplicities, $\prec$) *where*

1. CLASS $\subseteq \mathcal{N}$ *is a set of names representing a set of classes, where* $\mathcal{N} \subseteq \mathcal{A}^+$ *is a non-empty set of names over alphabet* $\mathcal{A}$. *Each class* $c \in$ CLASS *induces an object type* $t_c \in T$. *The values of the type refer to the objects of the class.*
2. ATT_c *is the attributes of a class* $c \in$ CLASS, *defined as a set of signatures* $a : t_c \rightarrow t$, *where the attribute name* a *is an element of* $\mathcal{N}$, $t_c \in T$ *is the type of class* c, *and* $t \in T$ *is the type of the attribute.*
3. OP_c *is a set of signatures for user-defined operations of a class* c *with type* $t_c \in T$. *The signatures are of the form* $\omega : t_c \times t_1 \times ... \times t_n \rightarrow t$, *where* ω *is the name of the operation, and* $t, t_1, ..., t_n$ *are types in* T.
4. ASSOC *is a set of association names.*
   (a) associates : ASSOC $\rightarrow$ CLASS^+ *is a function mapping each association name to a list of participating classes. This list has at least two elements.*
   (b) roles : ASSOC $\rightarrow$ $\mathcal{N}^+$ *is a function mapping each association to a list of role names. It assigns each class participating in an association a unique role name.*
   (c) multiplicities : ASSOC $\rightarrow$ $\mathcal{P}(\mathbb{N}_0)^+$ *is a function mapping each association to a list of multiplicities. It assigns each class participating in an association a multiplicity. A multiplicity is a non-empty set of natural numbers (an element of the power set* $\mathcal{P}(\mathbb{N}_0)^+$*) different from* $\{0\}$.
5. $\prec$ *is a partial order on* CLASS *reflecting the generalization hierarchy of classes.*

Figure 5 visualizes an object model in the form of a UML class diagram.

An interpretation of an object model is referred to as a snapshot (a system state). A snapshot is constituted by objects, links, and attribute values.

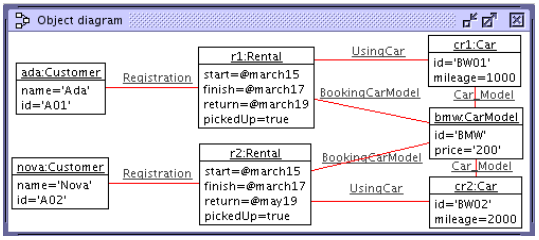


Fig. 6: A snapshot visualized by an object diagram.

Definition 6 (Snapshots). *A snapshot of an object model* $\mathcal{M}$ *is the structure*
$\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}})$ *such that:*

1. *For each* $c \in$ CLASS, *the finite set* $\sigma_{\text{CLASS}}(c)$ *contains all objects of class* $c \in$ CLASS *existing in the snapshot:* $\sigma_{\text{CLASS}}(c) \subset oid(c)$.
2. *Functions* σ_{ATT} *assign attribute values for each object in the state.* $\sigma_{\text{ATT}}(a)$: $\text{CLASS}(c) \rightarrow I(t)$ *for each* $a : t_c \rightarrow \text{ATT}^*_c$.
3. *For each* $as \in$ ASSOC, *there is a set of current links:* $\sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$. *A link set must satisfy all multiplicity specifications:* $\forall i \in \{1, ..., n\}, \forall l \in \sigma_{\text{ASSOC}}(as)$: $|\{l' | l' \in \sigma_{\text{ASSOC}}(as) \wedge (\pi_i(l') = \pi_i(l))\}| \in \pi_i(\text{multiplicities}(as))$

where

- $I(t)$ *is the domain of each type* $t \in T$.
- $oid(c)$ *is the objects of each* $c \in$ CLASS. *The set is often infinite.* $I_{\text{CLASS}}(c) = oid(c) \cup \{oid(c') | c' \in \text{CLASS} \wedge c' \prec c\}$.
- ATT^*_c *is the direct and inherited attributes of the class* c: $\text{ATT}^*_c = \text{ATT}_c \cup_{c<c'} \text{ATT}_{c'}$.
- $I_{\text{ASSOC}}(as) = I_{\text{CLASS}}(c_1) \times ... \times I_{\text{CLASS}}(c_n)$ *interprets the association as, where* associations$(as) = \langle c_1, c_2, ..., c_n \rangle$, $as \in$ ASSOC, *and* $c_1, c_2, ..., c_n$ *are the classes. Each* $l_{as} \in I_{\text{ASSOC}}(as)$ *is a link.*
- $\pi_i(l)$ *projects the ith component of a list* l.

Figure 6 visualizes a snapshot in the form of a UML object diagram. The snapshot is the interpretation of the object model shown in Fig. 5.

3.3. Models and Metamodels

A *model* is a representation in a certain medium of something in the same or another medium. The model captures the important aspects of the thing being modeled from a certain point of view and simplifies or omits the rest [12]. The medium to express models, a convenience for working, can be 3-D figures in a paper, a computer for models of buildings, or modeling languages. Our work focuses on modeling languages, defined using *metamodels*, as the means to express models.

Definition 7 (Metamodels). *A metamodel is the structure* MM = ($\mathcal{M}$, WFC) *where the* $\mathcal{M}$ *is an object model and the* WFC *is a set of OCL conditions, so-called well-formedness conditions, w.r.t the OCL algebra built on the* $\mathcal{M}$.

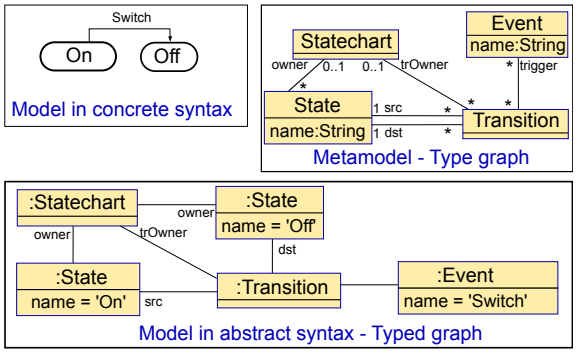


Fig. 7: The simplified metamodel for statechart models.

Figure 7 shows a simplified metamodel for statecharts. The graph corresponding to the metamodel is referred to as a type graph. The object model $\mathcal{M}$ includes 4 classes (*Statechart, State, Transition, and Event*) and 5 associations. The *WFC* includes the invariant `ownsChildState`, *"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)
```

Definition 8 (Models). *Let a metamodel* MM = ($\mathcal{M}$, WFC) *be given. A model that conforms to the metamodel* MM *is a snapshot of the* $\mathcal{M}$ *and the snapshot fulfills all the OCL invariants of the* WFC.

Figure 7 shows a model that conforms to the statechart metamodel. The graph corresponding to the model is referred to as a typed graph. The model might be represented in different forms, i.e., in abstract syntax or concrete syntax.

3.4. Incorporation of OCL and Triple Rules

Within the context where the underlying type graph represents a metamodel, we could employ OCL conditions in order to restrict the applicability of triple rules. The aim is 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 $s.name <> null \wedge aut1.name <> null$. We could define OCL application conditions for triple rules as follows.

Definition 9 (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 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 the BAC_{xy} *with* $xy \in \{'SL', 'SR', 'CL', 'CR', 'TL', 'TR'\}$ *are the BACs in the LHS and RHS of the source, correspondence, and target parts of the triple rule, respectively; the* BAC_{pre} *and* BAC_{post} *are the pre- and postconditions, respectively.*

[1]BACs stands for Boolean Application Conditions

Definition 10 (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 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.*

Definition 11 (Restrictions on Derived Rules). *Let a triple rule tr be given. The preconditions of its derived triple rules 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*
- $BAC^{trI}_{pre} = BAC^{tr}_{SR*} \cup BAC^{tr}_{CL} \cup BAC^{tr}_{TR*}$

The postconditions 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 postconditions,*
- BAC^{tr}_{xy} *with* $xy \in \{'SL', 'SR', 'CL', 'CR', 'TL', 'TR'\}$ *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.*

3.5. Model Transformations

The aim of TGGs is to ease the description of complex transformations. Structural mappings within triple rules allow us to relate the source, target, and correspondence parts within a triple derivation: Once a plain rule derivation for the source (or target) model is given, we can induce two derivations corresponding to the remaining parts. In this way operational scenarios of triple rules for model transformations are defined.

Let $TGG = (TG, S, TR)$ be a triple graph grammar incorporating OCL. Let 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 12 (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 13 (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 14 (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 15 (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* $(E_S \leftarrow E_C \rightarrow E_T) \overset{*}{\Rightarrow} (F_S \leftarrow F_C \rightarrow F_T)$.

Theorem 1 (Derived Rules for Transformations). *Let* $TGG = (TG, S, TR)$ *be a triple graph grammar incorporating OCL and* $(G_S \leftarrow S_C \rightarrow S_T)$ *be a triple graph typed by TG.*

We could obtain the forward transformation from G_S *to* G_T*, i.e.,* $S \overset{*}{\Rightarrow} (G_S \leftarrow G_C \rightarrow G_T)$*, 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 (i), 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 of the transformation in (i), we have $(G_S \leftarrow G^i_C \rightarrow G^i_T) \overset{trF_{i+1},m_{i+1}}{\Longrightarrow} (G_S \leftarrow G^{i+1}_C \rightarrow G^{i+1}_T)$. Then, 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 backward and integration transformation, we can obtain a similar result. The condition (ii) in these cases is shown respectively as follow.

$(S_S \leftarrow S_C \rightarrow G_T) \overset{trB_1,m_1}{\Longrightarrow} \ldots \overset{trB_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$,

$(G_S \leftarrow S_C \rightarrow G_T) \overset{trI_1,m_1}{\Longrightarrow} \ldots \overset{trI_n,m_n}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$.

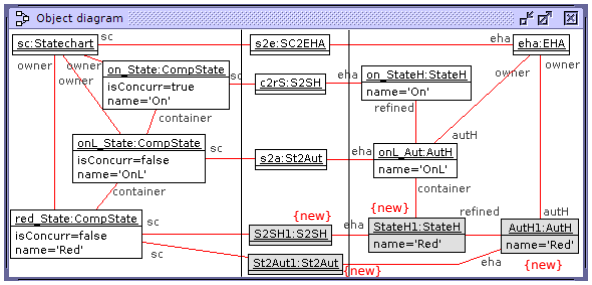


Fig. 8: A forward transformation step by the forward rule derived from the rule shown in Fig. 4.

Figure 8 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.

4. A Transformation Model in OCL

This section focuses on the operational scenarios derived from triple rules including OCL. We employ OCL in order to realize the operational scenarios of triple rules towards an OCL-based framework for model transformation. The OCL framework also offers a new operation for model synchronization.

4.1. Basic Idea

As illustrated in Fig. 9, we consider each transformation scenario derived from a triple rule as a special kind of *model behavior*, namely behavior of operations. Each transformation scenario can be mapped to an operation. We realize the operations by taking two views on them: Declarative OCL pre- and postconditions are employed as operation contracts, and imperative OCL command sequences are taken as an operational realization.

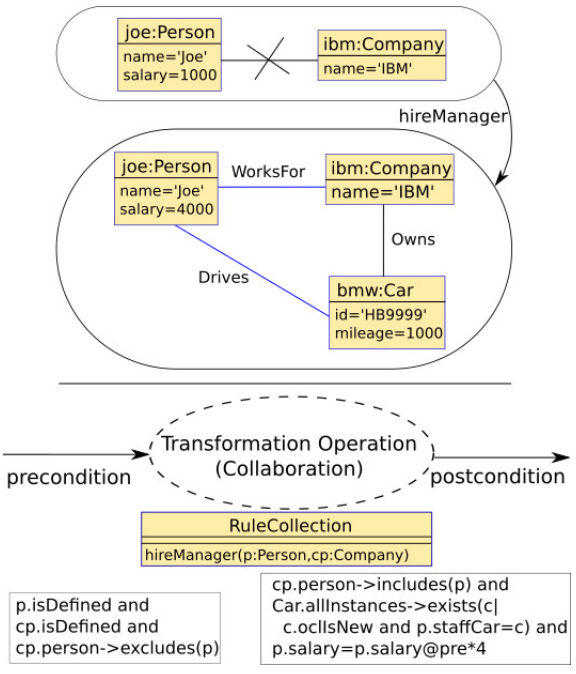


Fig. 9: Illustration for an OCL transformation.

4.2. OCL Transformation Operations

Figure 10 depicts the input of transformation operations derived from triple rules. We use a sheet including six cells that correspond to six patterns of the original triple rule in order to describe the input of each operation. The cell denoted by 'I' means that nodes in this part belong to the input of the operation. The cell denoted by '?' represents objects created by the operation. The cell denoted by 'U' means that part of this cell belongs to the input of the operation, and this part can be updated by the operation. The remaining nodes in this cell correspond to objects created by the operation.

Operational Scenarios	Input/Computing
Model Co–Evolution	I I I / ? ? ?
Forward Transformation	I I I / I ? ?
Model Integration	I I I / I ? I
Model Synchronization	I I I / U I U

I: Input
?: Create
U: Update

SL	CL	TL
SR	CR	TR

maskS=SR\SL
maskC=CR\CL
maskT=TR\TL

Fig. 10: The input and computation for derived triple rules.

```
RuleCollection
-------------------------------------Model Co-Evolution
compStateNest_coEvol(
  matchSL:Tuple(sc:Statechart,cps:CompState,_s_name:String),
  matchTL:Tuple(eha:EHA,aut:AutH,_aut1_name:String),
  matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut))
-------------------------------------Forward Transformation
compStateNest_forwTrafo(
  matchSR:Tuple(s:CompState,sc:Statechart,cps:CompState),
  matchTL:Tuple(eha:EHA,aut:AutH,_aut1_name:String),
  matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut))
-------------------------------------Model Integration
compStateNest_integraTrafo(
  matchSR:Tuple(s:CompState,sc:Statechart,cps:CompState),
  matchTR:Tuple(sH:StateH,aut1:AutH,eha:EHA,aut:AutH),
  matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut))
-------------------------------------Model Synchronization
compStateNest_synchTrafo(
  matchSL:Tuple(sc:Statechart,cps:CompState,_s_name:String),
  matchTL:Tuple(eha:EHA,aut:AutH,_aut1_name:String),
  matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut),
  maskS:Tuple(s:CompState),
  maskT:Tuple(sH:StateH,aut1:AutH),
  maskC:Tuple(s2sH:S2SH,s2a1:St2Aut))
```

Fig. 11: Transformation operations derived from the triple rule `compStateNest` shown in Fig. 4.

Figure 11 presents the derived operations w.r.t the triple rule depicted in Fig. 4. Note that when an entry of a mask parameter (`maskS`, `maskT`, or `maskC` having the `Tuple` type) in a synchronization operation is undefined, a new object corresponding to this entry is newly created. Otherwise, the entry will be updated.

4.3. Example Transformation Scenarios

We have defined 9 triple rules for the SC2EHA transformation as summarized in Table 1. We illustrate transformation scenarios by explaining informally a transformation step of each scenario. These example transformation steps are performed by operations derived from the triple rule shown in Fig. 4.

4.3.1. Forward Transformation

The transformation step for the forward transformation from the statechart to the EHA is illustrated as shown in Fig. 8. The match for this application includes objects highlighted in the first object diagram. The part (objects, nodes) which is newly created includes objects highlighted in the second object diagram. This means that all nodes and links in the source side of the original rule (Fig. 4) are used for matching the derived triple rule.

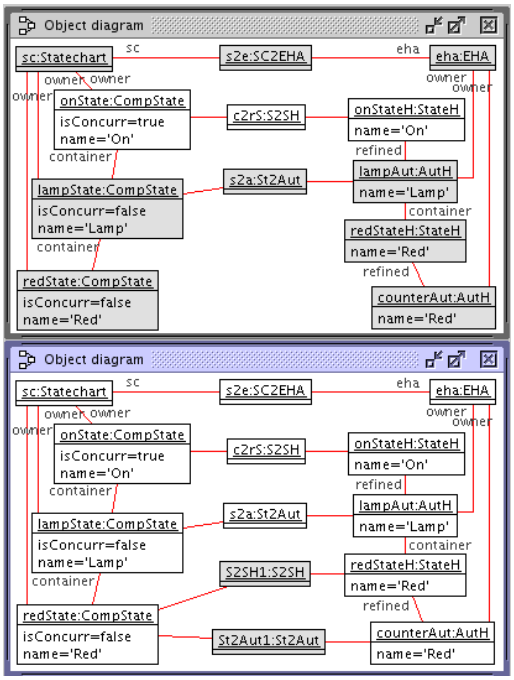


Fig. 12: Example model integration step.

Table 1: Triple rules for the SC2EHA transformation

Triple rule	Effect on statechart	Effect on EHA
initTop	to create the initial state, a simple state, and the transition between them, e.g., to create the `Off` state.	to create the top state, the top automaton, and its initial state, e.g., to create the initial state `Off`.
initNest	to create the initial pseudo state, a simple state, and the transition between them, e.g., to create the state `Green`.	to create the initial state of the non-top automaton, e.g., to create the initial state `Green`.
simpStateNest	to create a simple state as the child of a composite state, e.g., to create the state `Yellow` or `Count1`.	to create a simple state as the child of an automaton, e.g., to create the state `Yellow` or `Count1`.
concurrStateTop	to create a concurrent state at the top level, e.g., to create the `On` state.	to create a state (e.g., the `On` state) of the top automaton and the automatons corresponding to the regions of the composite state. These automatons refine the state.
compStateNest	to create a non-concurrent composite state as a child of a composite state, e.g., to create the `Red` state.	to create a state and an automaton refining the state, e.g., to create the `Red` state and the automaton refining this state.
transitToSimp	to create a non-interlevel transition to a simple state, e.g., to create the transition from `Green` to `Yellow`.	to create a transition in an automaton, e.g., to create the transition from `Green` to `Yellow`.
transitToConcurr	to create a non-interlevel transition to a concurrent state, e.g., to create the transition from `Off` to `On`.	to create a transition to a state that is refined by automatons. This transition is labeled with the target determination set. For example, to create the transition from `Off` to `On`.
transitToComp	to create a non-interlevel transition to a non-concurrent composite state, e.g., to create the transition from `Yellow` to `Red`.	to create a transition together with labels for the target determination set, e.g., to create the transition from `Yellow` to `Red` and its target determination set.
transitUpSimp	to create an interlevel transition up to a state at the next level, e.g., to create the transition from `Count2` to `RedYellow`.	to create a transition from a state refined by automatons and its source restriction set, e.g., to create the transition from `Red` to `RedYellow` and its source restriction set.

4.3.2. Model Integration

The transformation step to integrate the example statechart and the example EHA is shown as in Fig. 12. We recognize that new parts occur only in the correspondence part. The match for this application includes all nodes and links in the source and target sides of the original rule.

4.3.3. Model Synchronization

Figure 13 shows the transformation step to integrate the statechart and the EHA. The effect of the so-called synchronization step includes: (1) two objects highlighted in the second object diagram are created, and (2) the `name` attribute of the `redStateH` object is updated.

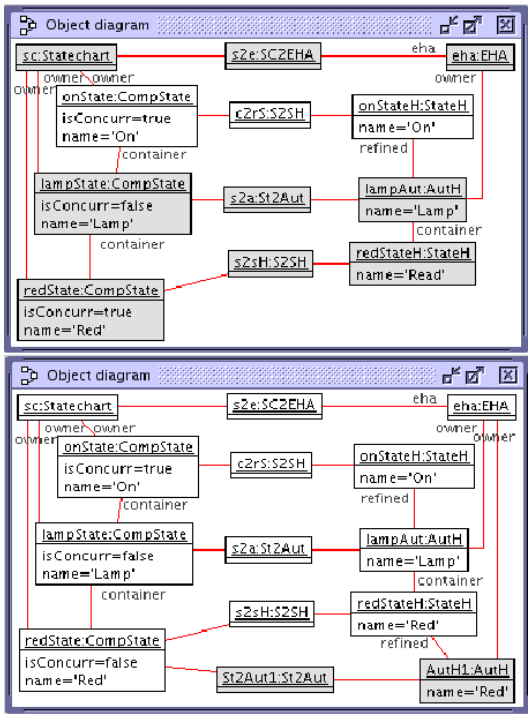


Fig. 13: Example model synchronization step.

5. Quality Assurance of Transformations

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

5.1. Verification of Transformation

We explain it in a formal way: Let MM_S and MM_T be metamodels for source and target models, respectively. Let $TGTS = (TG, S, TR)$ be a TGG, which relates source and target models to each other. Forward operations allows us to define a corresponding target model M_T for each source model M_S. We need to check if the target model M_T is correctly defined. Note that the verification for other transformation scenarios can be similarly obtained.

5.1.1. Check Invariants of Transformations

Triple rules can be viewed 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 to represent the pattern of this rule. For example, the following OCL invariant of the *CompStateNest* class represents the rule *CompStateNest* shown in Fig. 4. The transformation is correct only if such an invariant are valid.

```
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)
```

5.1.2. Check Contract Fulfillment

A forward transformation is realized as a sequence of OCL operation applications: $d_{tr} : (M_S \leftarrow \phi \rightarrow \phi) \overset{trF_1,m_1}{\Longrightarrow} \ldots \overset{trF_n,m_n}{\Longrightarrow} (M_S \leftarrow M_C \rightarrow M_T)$, 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.

5.1.3. 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')
```

5.1.4. 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)
```

5.2. Validation of Transformation

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

5.2.1. 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.

5.2.2. 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.

6. The RTL Language and 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. 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 illustrated in Fig. 14.

With the full OCL support, USE allows us to realize transformations and to ensure their correctness as discussed in Sect. 5: We could

[2]RTL stands for Restricted Graph Transformation Language

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 transformation framework based on the integration of TGGs and OCL are completely covered by USE. Figure 15 shows metamodels for the SC2EHA transformation in USE.

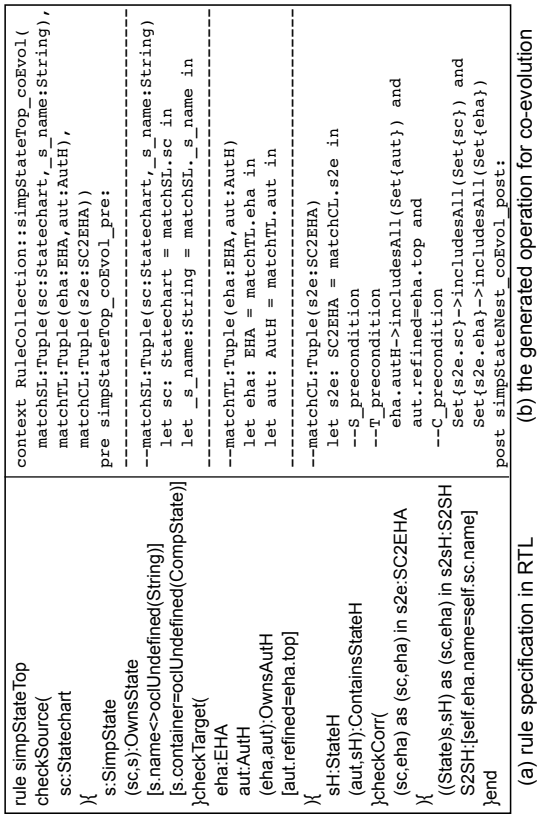


Fig. 14: RTL specification and generated OCL operations.

7. Related Work

Triple Graph Grammars (TGGs) have been proposed in [8]. Since then, many works have extended TGGs for software engineering [17]. Here we focus on the incorporation of TGGs and OCL as a foundation for transformations as proposed in our previous work [18, 19, 20]. This work is an extended version of our previous work with a focus on a formal foundation and an OCL-based framework for model transformations.

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 [17]. Many other works focus on the translation of the transformation to a formal domain for model checking such as Alloy in [21], Promela in [22], and Maude in [23].

In the field of Model-Driven Engineering, testing and analysis of model transformations has been subject to investigations (see, for example, [24, 25]). The work [26] proposes a technique for developing test cases for UML and OCL models. By guiding the construction process through so-called classifying terms, the built test cases in form of object models are classified into equivalence classes. 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 verification and validation of transformations. The proposal in [27] 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.

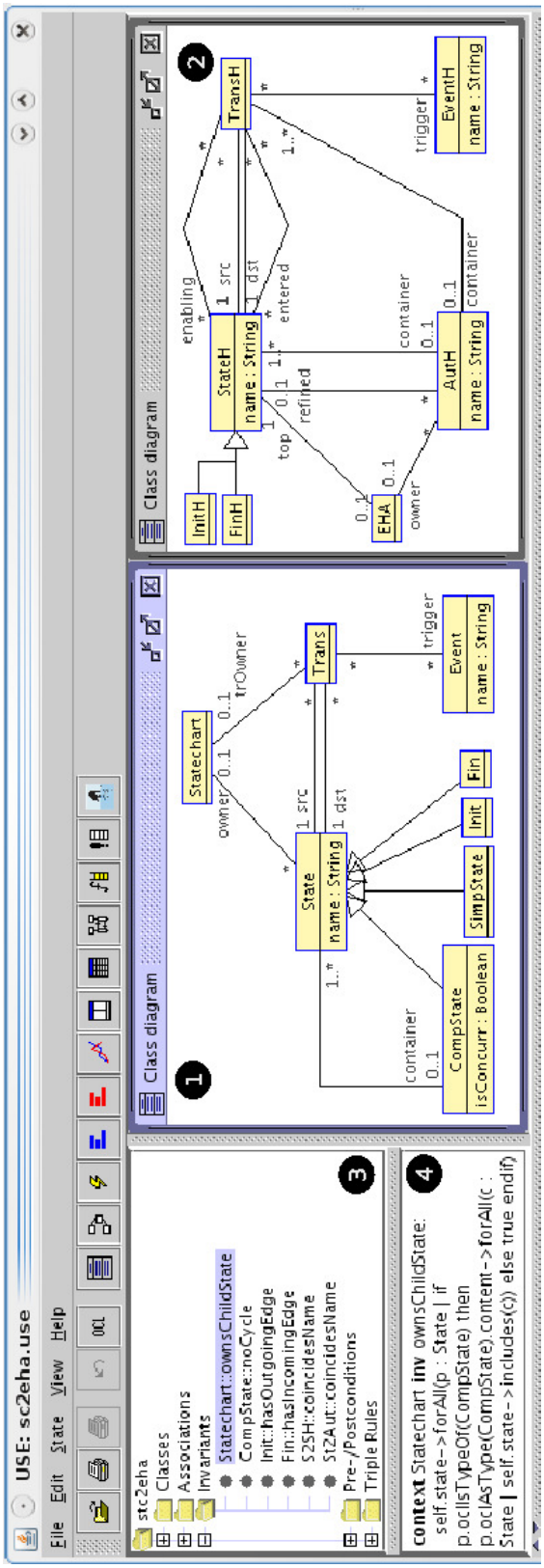


Fig. 15: Metamodels for the SC2EHA transformation.

8. Conclusion

We have introduced an approach for specifying, realizing, and ensuring the quality 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 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 RTL specification of transformations. This work also embodies a new method to extract invariants for transformations. The central idea is to view transformations as models. (3) An OCL-based framework for model transformation has been 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 QG.14.06, Vietnam National University, Hanoi. We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

[1] S. Sendall, W. Kozaczynski, Model Transformation: the Heart and Soul of Model-Driven Software Development, IEEE Software 20 (5) (2003) 42– 45.

[2] T. Mens, P. V. Gorp, A taxonomy of model transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125 – 142.

[3] F. Jouault, F. Allilaire, J. Bzivin, I. Kurtev, ATL: A Model Transformation Tool, Science of Computer Programming 72 (1-2) (2008) 31–39.

[4] P.-A. Muller, F. Fleurey, J.-M. Jzquel, Weaving Executability into Object-Oriented Meta-languages, in: Proc. 8th. Int. Conf. Model Driven Engineering Languages and Systems (MODELS), 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, H. Charaf, Validated Model Transformation-Driven Software Development, Int. J. Comput. Appl. Technol. 31 (1/2) (2008) 106–119.

[7] D. Varr, A. Pataricza, Automated Formal Verification of Model Transformations, in: J. Jrjens, B. Rumpe, R. France, E. B. Fernandez (Eds.), Proc. 3rd Workshop on Critical Systems Development in UML (UML/CSDUML), Technische Universitt Mnchen, 2003, pp. 63–78.

[8] A. Schrr, Specification of Graph Translators with Triple Graph Grammars, in: M. Schmidt (Ed.), Proc. 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG), Vol. 903 of LNCS, Springer-Verlag, 1995, pp. 151–163.

[9] J. Cabot, R. Claris, E. Guerra, J. d. Lara, Verification and Validation of Declarative Model-to-model Transformations Through Invariants, Journal of Systems and Software 83 (2) (2010) 283–302.

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

[11] M. Gogolla, F. Büttner, M. Richters, USE: A UML-Based Specification Environment for Validating UML and OCL, Science of Computer Programming 69 (1-3) (2007) 27–34.

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

[13] E. Mikk, Y. Lakhnechi, M. Siegel, Hierarchical automata as model for statecharts, in: Advances in Computing Science, Vol. 1345, Springer Berlin, 1997, pp. 181–196.

[14] G. Pintr, I. Majzik, Modeling and Analysis of Exception Handling by Using UML Statecharts, in: Scientific Engineering of Distributed Java Applications, Vol. 3409, LNCS, 2005, pp. 58–67.

[15] J. B. Warmer, A. G. Kleppe, The Object Constraint Language: Precise Modeling with UML, 1st Edition, Addison-Wesley Professional, 1998.

[16] M. Richters, A Precise Approach to Validating UML Models and OCL Constraints, Ph.D. thesis, Universitt Bremen, Fachbereich Mathematik und Informatik (2002).

[17] J. Greenyer, E. Kindler, Reconciling TGGs with QVT, in: G. Engels, B. Opdyke, D. C. Schmidt, F. Weil (Eds.), Proc. 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS), Vol. 4735 of LNCS, Springer, 2007, pp. 16–30.

[18] D. Dang, M. Gogolla, An approach for quality assurance of model transformations, in: D. V. Hung, H. T. Vo, J. Sanders, L. T. Bui, S. B. Pham (Eds.), Proc. 4th Int. Conf. Knowledge and Systems Engineering (KSE), IEEE Computer Society, 2012, pp. 223–230.

[19] D.-H. Dang, M. Gogolla, Precise Model-Driven Transformation Based on Graphs and Metamodels, in: D. V. Hung, P. Krishnan (Eds.), Proc. 7th Int. Conf. Software Engineering and Formal Methods (SEFM), IEEE Computer Society Press, 2009, pp. 307–316.

[20] D.-H. Dang, M. Gogolla, On Integrating OCL and Triple Graph Grammars, in: M. Chaudron (Ed.), Models in Software Engineering, Workshops and Symposia at MODELS. Reports and Revised Selected Papers, Vol. 5421, Springer, 2009, pp. 124–137.

[21] K. Anastasakis, B. Bordbar, J. M. Kster, Analysis of Model Transformations via Alloy, in: Proc. 4th Workshop on Model-Driven Engineering, Verification and and Validation (MoDeVVA), 2007, pp. 47–56.

[22] D. Varr, Automated Formal Verification of Visual Modeling Languages by Model Checking, Software and Systems Modeling 3 (2) (2004) 85–113.

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

[24] L. A. Rahim, J. Whittle, A survey of approaches for verifying model transformations, Software and System Modeling 14 (2) (2015) 1003–1028.

[25] G. M. K. Selim, J. R. Cordy, J. Dingel, Model transformation testing: The state of the art, in: Proc. 1st Workshop on the Analysis of Model Transformations, ACM, 2012, pp. 21–26.

[26] M. Gogolla, A. Vallecillo, L. Burgueño, F. Hilken, Employing classifying terms for testing model transformations, in: T. Lethbridge, J. Cabot, A. Egyed (Eds.), Proc. 18th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS), IEEE, 2015, pp. 312–321.

[27] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner, Towards Verified Model Transformations, in: D. Hearnden, J. G. S, B. Baudry, N. Rapin (Eds.), Proc. 3rd Int. Workshop on Model Development, Validation and Verification (MoDeVVA), Le Commissariat l'Energie Atomique - CEA, 2006, pp. 78–93.