Precise Model-Driven Transformations Based on Graphs and Metamodels

Duc-Hanh Dang and Martin Gogolla
Department of Computer Science, University of Bremen,
D-28334 Bremen, Germany
{hanhdd|gogolla}@informatik.uni-bremen.de

***Abstract*—Presenting precisely models and supporting automatic manipulation of models are at the heart of model-centric software development. A formal foundation for these tasks is a necessity as well as a challenge. We present a model-driven approach based on the integration of two light-weight formal methods, the Object Constraint Language (OCL) and Triple Graph Grammars (TGGs). OCL together with metamodeling allows us to present precisely models. With TGGs we can carry out model manipulations, especially model transformations. We focus on explaining the tool which realizes our approach. This tool is developed as an extension of the UML-based Specification Environment (USE), which offers full OCL support. A case study showing the transformation between statecharts and extended hierarchical automata explains our approach.**

***Keywords*-Model-Driven Transformation, Object Constraint Language (OCL), Graph Transformation, Invariant, Pre- and Postcondition.**

I. Introduction

Software systems become more and more complex. Despite significant advances in code-centric technologies such as advanced programming languages and integrated development environments (IDEs), developing complex software systems is still a laborious task. Model-centric software development has emerged as a promising paradigm, indicating a shift from "code-centric" to "model-centric" engineering. This paradigm puts forward a necessity as well as a challenge of a formal foundation for presenting precisely models and supporting automatic model manipulations.

Metamodeling and model transformation are currently the backbone of model-centric software engineering. On the one hand, metamodeling, which is a means for defining modeling languages, aims to precisely present models for specific purposes. On the other hand, model transformation is a robust approach for (1) transforming models from one language into anther language, (2) tackling the challenge *how to choose a right level of abstraction, and to relate levels of abstraction with each other* [1], and (3) supporting software evolution and maintenance.

Many approaches to model transformation have been introduced. ATL [2] and Kermeta [3] are well-known systems supporting transformation languages. They aim to realize the Query/View/Transformation (QVT) [4] standard for model transformation, which is proposed by the Object Management Group (OMG). Another promising approach for model transformation is the one based on graph transformation. VMTS [5], Fujaba [6], and VIATRA [7] are among many tools realizing this approach for unidirectional model transformation. Triple Graph Grammars (TGGs), introduced in [8], tend to be a promising approach for explaining relationships between models, especially, bidirectional model transformations. Several tools such as MOFLON [9] and AToM3 [10] implement TGGs for model transformation.

In this paper we present a model-driven approach based on the integration of two light-weight formal methods, the Object Constraint Language (OCL) and TGGs. We realize this approach by a tool as an extension of USE (UML-based Specification Environment) [11]. The advantages of our approach include two main points. First, the integration of TGGs and OCL allows us to increase the expressiveness of transformation rules. The USE4TGG language is able to present and implement triple rules including OCL. Second, our OCL-based approach allows us to operationalize triple rules including OCL. This results in an on-the-fly verification of model transformations and means for transformation quality assurance, i.e., checking well-formedness of models and maintaining model consistency.

In order to illustrate our approach we present a case study showing the transformation between statecharts and extended hierarchical automata. This is an example transformation. Our approach is applicable for transformations between arbitrary languages. For example, the following transformations have been supported by our tool: (1) The transformation from a UML subset to a Java subset as explained in [12], (2) the transformation from UML activity diagrams to Communicating Sequential Processes (CSPs) as mentioned in [13], (3) the transformation between logic languages, and (4) the transformation between use case and design models as proposed in [14].

The rest of this paper is structured as follows. Section 2 describes the case study. Section 3 explains how our tool supports well-formed models. Section 4 explains the way the integration of TGGs and OCL can support model consistency and traceability. Section 5 explains our OCL-based approach for operational scenarios derived from triple rules for model transformation. Section 6 comments on related work. The paper is closed with a conclusion and a discussion of future work.

II. Running Example

We illustrate our approach by explaining the transformation between statecharts and extended hierarchical automata. This transformation is studied in a context where the models are used to specify a traffic supervisor system for a crossing of a main road and a country road

(Fig. 1). This example is inspired by [15]. The lamp controller provides higher precedence to the main road in the following way: 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 waiting as usual. A camera allows the system to record cars running illegally in the crossing during the red signal.

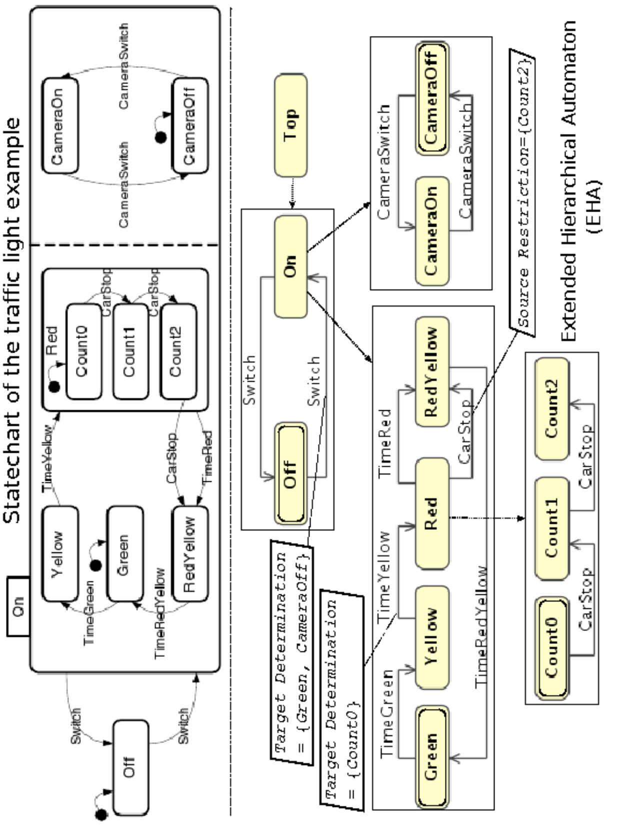

Figure 1. Statechart and extended hierarchical automaton for the traffic light example (adapted from [15])

A. Example Statechart

The system behavior is described by the UML statechart (state machine) [16] shown in Fig.1. The system has two basic *states* `On` and `Off`, reflecting whether the system is turned on or off. The `Off` state is the *initial state*. The system state is changed when the *event* `Switch` occurs. The response of the system in this case is a *transition* leaving from the current state (*source state*) to the *target state*. 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*. The `Red` state is also a composite state. The child states of this state indicate the number (0,1, or 2) of the waiting cars. They are simple states.

B. Example Extended Hierarchical Automaton

Extended Hierarchical Automata (EHAs) have been proposed in [17] 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 [17].

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.

C. Overview of the SC2EHA Transformation

We now describe informally the SC2EHA transformation between Statecharts and EHAs by showing intuitive correspondences between them. The correspondences are as follows.

1) **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.
2) **Simple state.** Each simple state of the statechart corresponds to a unique state of the sequential automaton.
3) **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 mentioned state of the sequential automaton.
4) **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 this state.

5) **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 as explained in Subsection II-B.

III. Well-Formed Models

We describe models in a modeling language based on the metamodeling approach. OCL conditions are used for restrictions on metamodels in order to precisely present models that are referred to as well-formed models.

A. *Metamodel Restrictions and Model Properties in OCL*

The core of the approach using OCL in connection with metamodeling in order to define models is based on the object-oriented paradigm. Specifically, the *conformsTo* relationship between a model and its metamodel is represented by the *instancesOf* relationship between an object diagram and a class diagram, respectively. Then, OCL conditions in a class diagram become restrictions on the corresponding metamodel. A model conforms to the metamodel only if these restrictions are fulfilled. In this way we can obtain a declarative definition of the modeling language.

Figure 2 pictures the simplified metamodel in USE for statecharts and EHAs. The class diagram marked by (1) presents meta-concepts for statecharts. These concepts are explained in Section II-A. Note that the `isConcurrent` attribute has a boolean value that specifies the decomposition semantics: If this attribute is true, then the composite state contains two or more orthogonal regions associated by a concurrent execution. Otherwise, there are no direct orthogonal region in the composite state.

The class diagram marked by (2) shows meta-concepts for EHAs. These concepts are also explained in Section II-B. This metamodel allows us to precisely present the example models shown in Fig. 1. The remaining concepts in this metamodel are used for the correspondence between statecharts and EHAs. These concepts are mentioned in Section IV.

1) Invariants: The window marked by (3) in Fig. 2 shows all invariants that are valid in a well-formed model. The invariant `ownsChildState` displayed in the class invariant window and marked by (4) 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)
```

2) 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

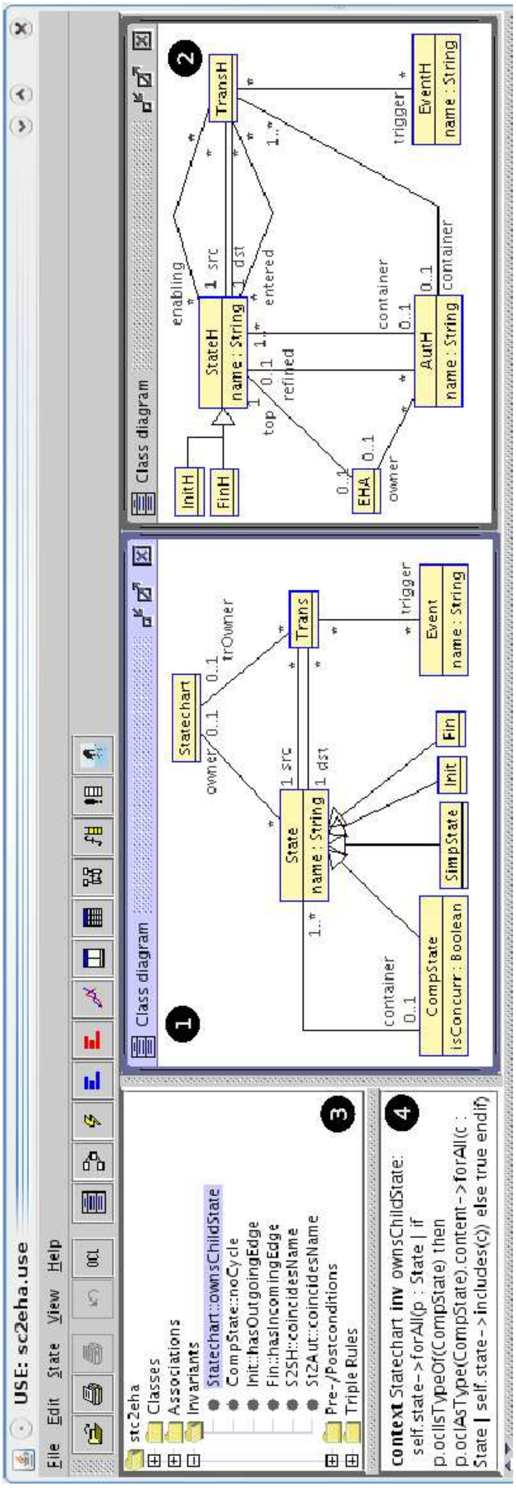

Figure 2. The metamodel in USE for Statecharts and EHAs

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')
```

IV. Model Consistency and Traceability

This section presents our approach based on the integration of TGGs and OCL [12] for explaining the relationship between models. The core of this approach is that models as instances of a metamodel can be operationally described by graph transformation rules in a corresponding graph grammar. We use TGGs including OCL to structurally map two graph grammars so that graphs produced by these graph grammars can be related to each other. Applying triple rules allows us to simultaneously build two models of the corresponding metamodels and to keep the correspondence between them. We refer to this process as model co-evolution.

A. Incorporating OCL in TGGs

Figure 3 shows a co-evolution step for building our example models. On the left side, the composite state `Red` is added into the `Lamp` state of the example statechart. On the right side, a corresponding change of the example EHA occurs.

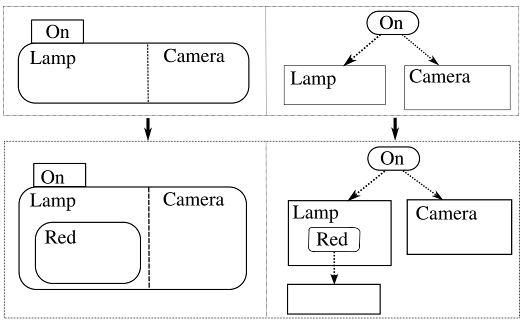

Figure 3. Co-evolution step for building the example models

We use triple rules including OCL in order to express such co-evolution steps. Figure 4 pictures the `compStateNest` triple rule for the step mentioned in Fig. 3. On the right top part, the graphical description of this rule is pictured. There are three *columns* in this description. The left and right columns correspond to the source (statechart) and target (EHA) models. The middle column describes the correspondence between the source and target models.

This rule can also be viewed considering the four *rows* in Fig. 4. The top and bottom rows display the OCL pre- and postconditions of the rule. The second row corresponds to the left-hand side of this rule. The nodes (objects) and edges (object links) of the left-hand side are determined when the rule is applied in a so-called *rule matching* process. The third row describes nodes and links that are created when the rule is applied. This part is the complement of the left-hand side (LHS) in the right-hand side (RHS) of the rule ($RHS \setminus LHS$).

On the left bottom part of Fig. 4, the textual description in the USE4TGG language [12] for triple rules (the `compStateNest` rule in this case) including OCL is shown. The USE4TGG description allows us to visualize triple rules and to generate the metamodel and invariants for the correspondence between the source and target models. The metamodel for the correspondence between statecharts and EHAs is shown in the class diagram, on the right bottom part of Fig. 4.

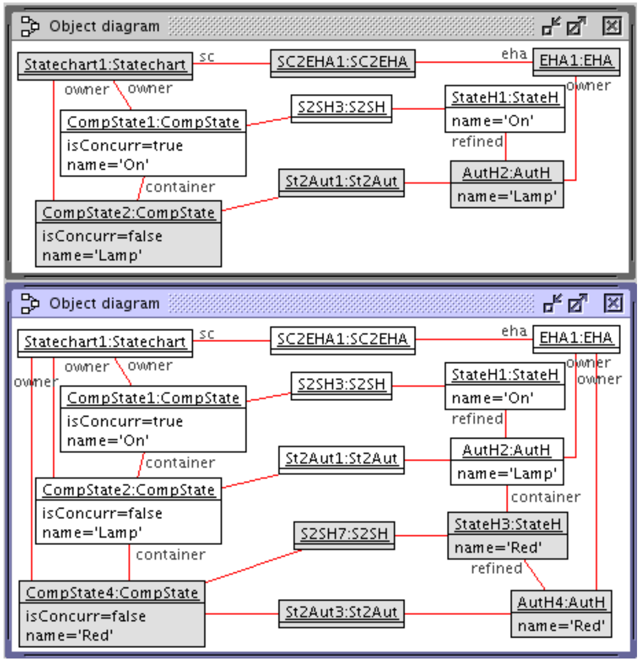

Figure 5. Co-evoluting model by triple rules including OCL

Figure 5 illustrates the application of triple rules, the `compStateNest` rule in this case, for model co-evolution. This transformation step corresponds to the one shown in Fig. 3. The first and second object diagrams present the working graph (the current transformation state) before and after the rule application, respectively. The highlighted part in the first object diagram corresponds to the left-hand side of this rule, indicating the result (*the match*) of the rule matching. The highlighted part in the second diagram, which corresponds to the part $RHS \setminus LHS$, includes all new objects and links created by the rule application.

B. Triple Rules for the SC2EHA Transformation

We define triple rules including OCL in order to simultaneously build the statechart and the corresponding EHA (co-evolution). Triple rules for our case study are summarized as shown in Table I.

Due to the limited space of this paper, one graphical description is shown in Fig. 4, the remaining rules for the example transformation are depicted in its long version[1].

[1]www.db.informatik.uni-bremen.de/publications/sefm09_long.pdf

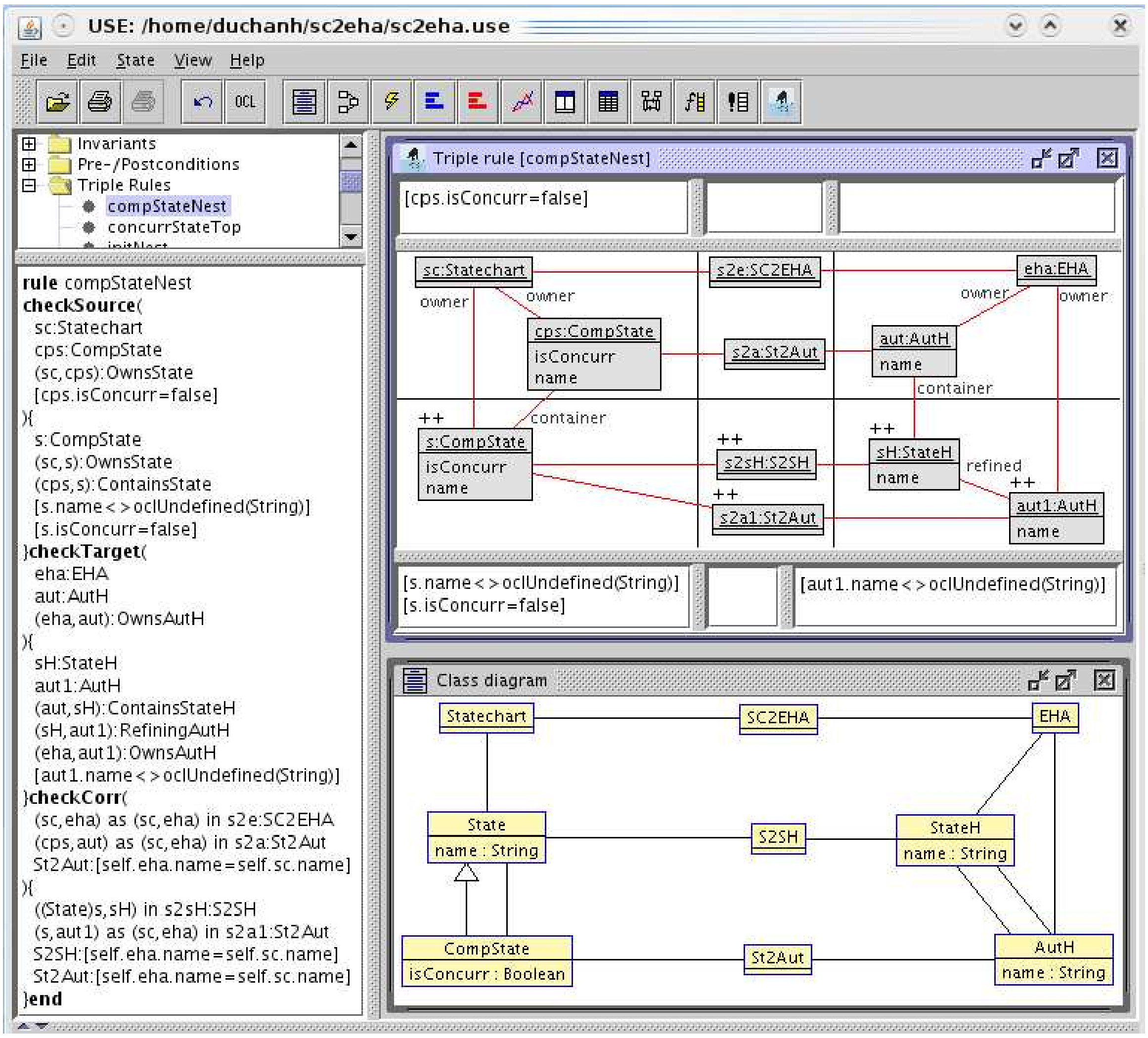

Figure 4. Presenting triple rules including OCL in USE

Table I
TRIPLE RULES FOR THE SC2EHA TRANSFORMATION IN THE CASE STUDY.

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 automata corresponding to the regions of the composite state. These automata 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 automata. 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 automata and its source restriction set, e.g., to create the transition from `Red` to `RedYellow` and its source restriction set.

C. *Model Co-Evolution for an Integrated Model*

Figure 6 shows the working graph at a transformation step when we apply triple rules mentioned above for a SC2EHA integrated model. At the final step of this model co-evolution we obtain the models as shown in Fig. 1 together with the correspondence between them. This correspondence is highlighted as pictured in Fig. 6.

Detecting model inconsistency. The correspondence between the source and target models within an integrated model produced by triple rules is represented by correspondence links (traces) and OCL constraints (including invariants). We can detect semi-automatically model inconsistency using OCL queries. For example, in Fig. 6 we perform the OCL query in order to check whether the names of two corresponding states coincide with each other.

V. Model Transformation

This section focuses on the operational scenarios derived from triple rules including OCL for forward- and backward transformation, model integration, and model synchronization. We use OCL in order to realize the operational scenarios of triple rules towards an OCL-based framework for model transformation.

A. *Example for Transformation Scenarios*

The following illustrates transformation scenarios by explaining informally a transformation step of each scenario. These example transformation steps are performed by operations derived from the `compStateNest` triple rule mentioned in Fig. 4 and Fig. 5.

1) Forward Transformation: The transformation step for the forward transformation from the statechart to the EHA is illustrated as shown in Fig. 7. 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.

2) Model Integration: The transformation step for the integration of the example statechart and the example EHA is pictured as shown in Fig. 8. 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.

3) Model Synchronization: Figure 9 shows the transformation step for the integration of the statechart and the EHA. The effect of the so-called model 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. The transformation operation offers the possibility to fix model inconsistencies that are mentioned in Section IV-C.

B. *Overview of Transformation Operations*

We consider each transformation scenario derived from a triple rule as a special kind of *model behavior*, namely

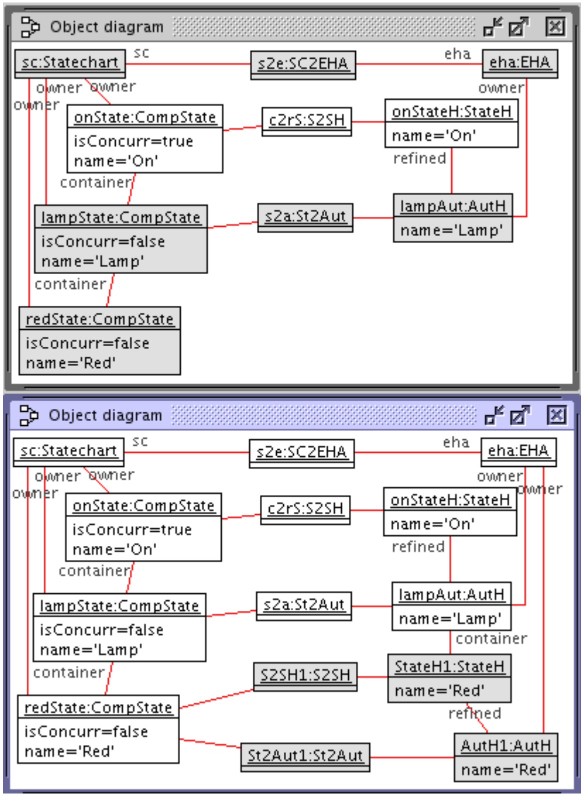

Figure 7. Forward transformation by triple rules including OCL

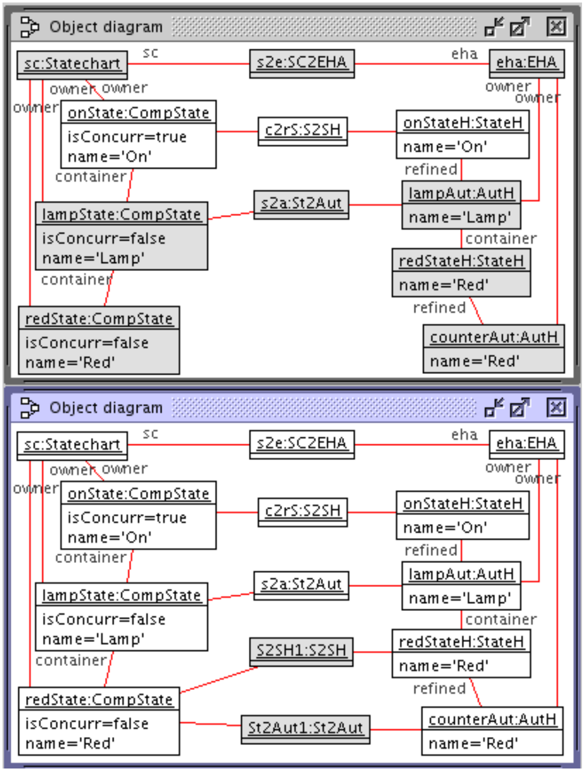

Figure 8. Model integration by triple rules including OCL

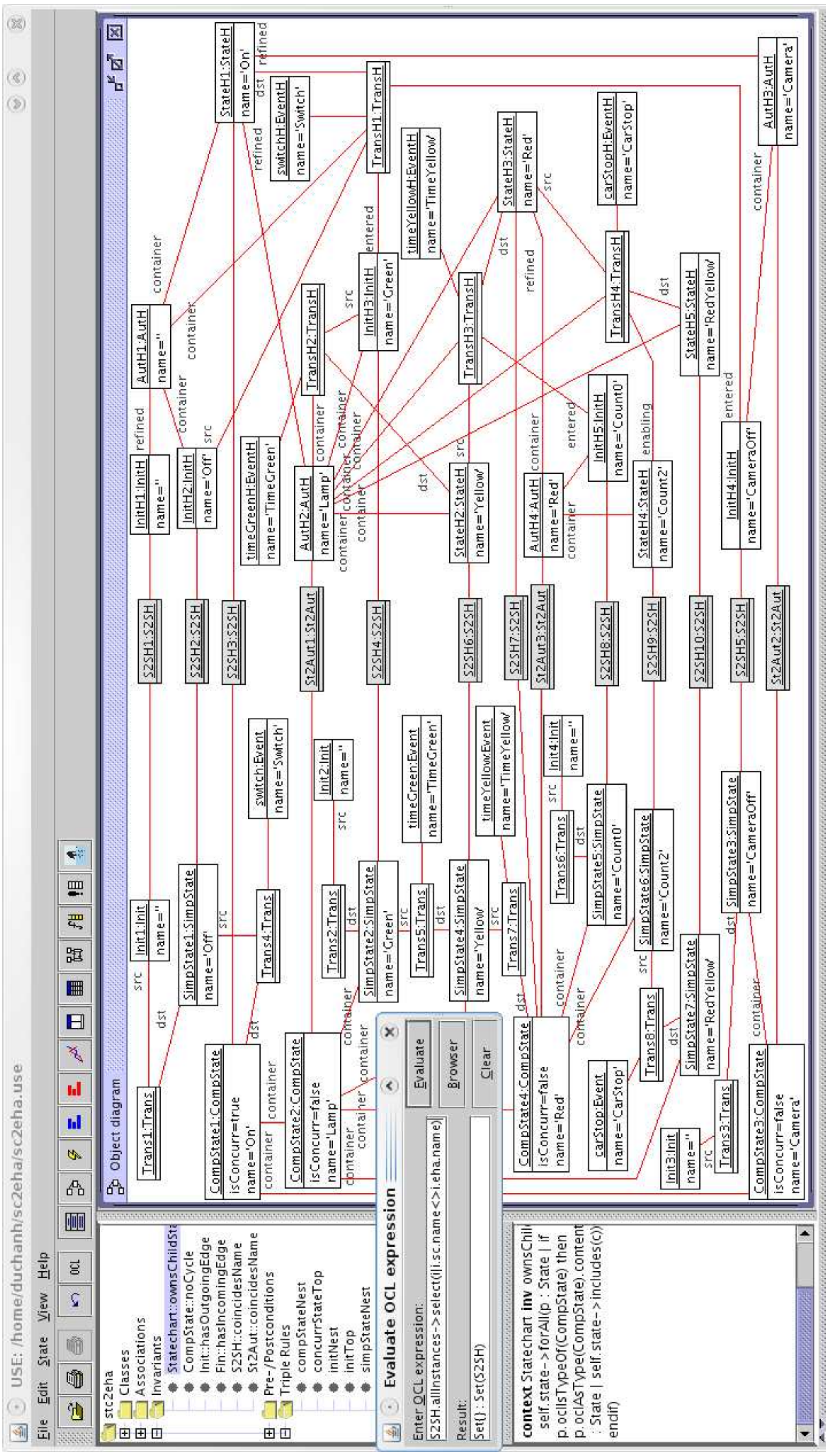

Figure 6. Model co-evolution by rule applications for the SC2EHA integrated model

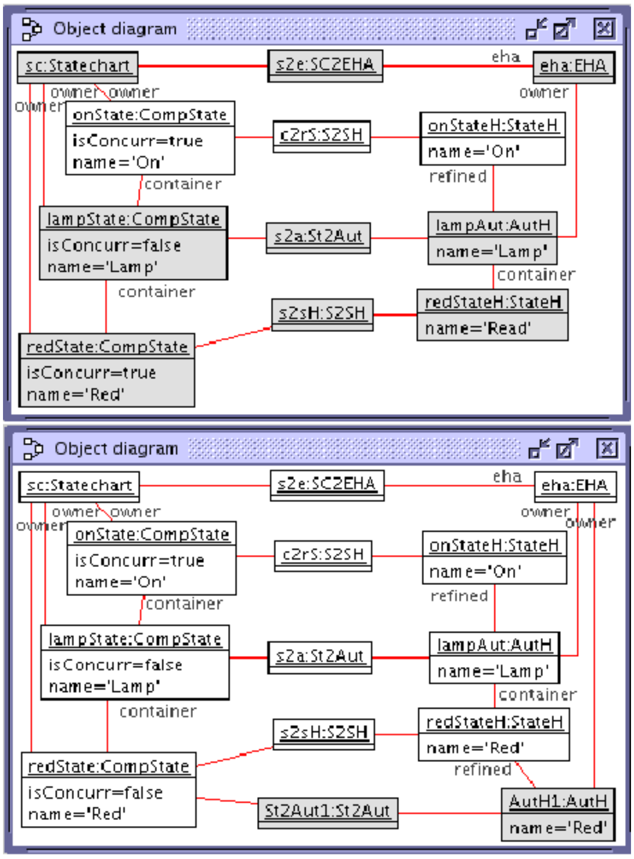

Figure 9. Model synchronization by triple rules including OCL

behavior of operations. This point is inspired by the work in [18], [3]. 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 USE command sequences are taken as an operational realization [12].

Operational Scenarios	Input/Computing	
Model Co–Evolution	I I I ? ? ?	I: Input ?: Create U: Update
Forward Transformation	I I I I ? ?	SL CL TL SR CR TR
Model Integration	I I I I ? I	maskS=SR\SL
Model Synchronization	I I I U I U	maskC=CR\CL maskT=TR\TL

Figure 10. The input and computation for derived triple rules

Figure 10 depicts the input of transformation operations derived from triple rules. The input of the operations is defined based on the way rules are matched as illustrated in Subsection V-A. We use a sheet including six cells that correspond to six patterns of the original triple rule as mentioned in Fig. 4 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.

```
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))
```

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

Figure 11 presents the derived operations of the triple rule depicted in Fig. 4. These derived operations are generated from the description in the USE4TGG language of the triple rule in a similar way as explained in [12]. 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, e.g., the `St2Aut1` object shown in Fig. 9, is newly created. In other case the value of this entry, e.g., the `s2SH` object, will be updated.

C. *Verification of Transformations*

Transformation operations derived from a triple rule can be characterized by pre- and postconditions. The precondition allows us to check if a rule is applicable. After each rule application, one may check the postcondition of the rule for an on-the-fly verification of the transformation.

Figure 12 and Fig. 13 depict the pre- and postcondition of the operation derived from the `compStateNest` triple rule for model co-evolution, respectively. These OCL conditions are generated from the USE4TGG description of the triple rule shown in Fig. 4 [12].

VI. Related Work

The transformation from statecharts to extended hierarchical automata is expressed first in [17]. That work and the work in [19], [20] give a formal description of the transformation. In this paper we present another approach to precisely specify the transformation.

The transformation in this paper can be seen as a transformation between two languages. This transformation kind is also the focus of the work in [13]. The work in [21] introduces a declarative approach to this transformation.

```
context RuleCollection::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))
pre compStateNest_coEvol_pre:
--matchSL:Tuple(sc:Statechart,cps:CompState,_s_name:String)
  let sc: Statechart = matchSL.sc in
  let cps: CompState = matchSL.cps in
  let _s_name:String = matchSL._s_name in
--matchTL:Tuple(eha:EHA,aut:AutH,_aut1_name:String)
  let eha: EHA = matchTL.eha in
  let aut: AutH = matchTL.aut in
  let _aut1_name:String = matchTL._aut1_name in
--matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut)
  let s2e: SC2EHA = matchCL.s2e in
  let s2a: St2Aut = matchCL.s2a in
  --S_precondition
    sc.state->includesAll(Set{cps}) and
    cps.isConcurr=false and
  --T_precondition
    eha.autH->includesAll(Set{aut}) and
  --C_precondition
    Set{s2e.sc}->includesAll(Set{sc}) and
    Set{s2e.eha}->includesAll(Set{eha}) and
    Set{s2a.sc}->includesAll(Set{cps}) and
    Set{s2a.eha}->includesAll(Set{aut})
```

Figure 12. OCL precondition of the operation derived from the triple rule shown in Fig. 4 for model co-evolution

```
context RuleCollection::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))
post compStateNest_coEvol_post:
--matchSL:Tuple(sc:Statechart,cps:CompState,_s_name:String)
  let sc: Statechart = matchSL.sc in
  let cps: CompState = matchSL.cps in
  let _s_name:String = matchSL._s_name in
--matchTL:Tuple(eha:EHA,aut:AutH,_aut1_name:String)
  let eha: EHA = matchTL.eha in
  let aut: AutH = matchTL.aut in
  let _aut1_name:String = matchTL._aut1_name in
--matchCL:Tuple(s2e:SC2EHA,s2a:St2Aut)
  let s2e: SC2EHA = matchCL.s2e in
  let s2a: St2Aut = matchCL.s2a in
  --S_postcondition
    Statechart.allInstances->includesAll(Set{sc}) and
    CompState.allInstances->includesAll(Set{cps}) and
    CompState.allInstances->exists( s | s.oclIsNew and
      sc.state->includesAll(Set{s,cps}) and
      cps.content->includesAll(Set{s}) and
      s.name<>oclUndefined(String) and
      s.isConcurr=false and
    --T_postcondition
      EHA.allInstances->includesAll(Set{eha}) and
      AutH.allInstances->includesAll(Set{aut}) and
      StateH.allInstances->exists( sH | sH.oclIsNew and
        AutH.allInstances->exists( aut1 | aut1.oclIsNew and
          Set{sH,aut1}->size = 2 and
          sH.refAut->includesAll(Set{aut1}) and
          eha.autH->includesAll(Set{aut1,aut}) and
          aut.state->includesAll(Set{sH}) and
          aut1.name<>oclUndefined(String) and
        --C_postcondition
          SC2EHA.allInstances->includesAll(Set{s2e}) and
          St2Aut.allInstances->includesAll(Set{s2a}) and
          S2SH.allInstances->exists( s2sH | s2sH.oclIsNew and
            St2Aut.allInstances->exists( s2a1 | s2a1.oclIsNew and
              Set{s2sH,s2a1}->size = 2 and
              Set{s2sH.sc}->includesAll(Set{s}) and
              Set{s2sH.eha}->includesAll(Set{sH}) and
              Set{s2a1.sc}->includesAll(Set{s}) and
              Set{s2a1.eha}->includesAll(Set{aut1}) and
              Set{s2e.sc}->includesAll(Set{sc}) and
              Set{s2e.eha}->includesAll(Set{eha}) and
              Set{s2a.sc}->includesAll(Set{cps}) and
              Set{s2a.eha}->includesAll(Set{aut}))))))
```

Figure 13. OCL postcondition of the operation derived from the triple rule shown in Fig. 4 for model co-evolution

Many approaches have been proposed for model transformation. Most of them focus on the standard QVT [4] for model transformation. ATL [2] and Kermeta [3] are well-known systems developed according to this standard. Like our work, they allow the developer to precisely present models using metamodels and OCL. Furthermore, transformation languages are used for unidirectional model transformation. 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 bidirectional model transformation.

Another promising approach for model transformation is the one based on graph transformation. VMTS [5], Fujaba [6], and VIATRA [7] are among many tools realizing this approach. They support unidirectional model transformation. Several tools based on TGGs [8] for bidirectional model transformation are introduced such as MOFLON [9] and AToM3 [10]. Like our work, these tools can achieve many QVT features since TGGs and QVT share many blocks and concepts [22]. The advantages of our approach include: (1) OCL pre- and postconditions allow us to increase the expressiveness of triple rules [12], (2) the USE4TGG language is proposed to present and implement triple rules including OCL, and (3) our approach offers an on-the-fly verification of model transformations for transformation quality assurance.

The formal foundation of the tool presented in this paper is based on the integration of TGGs and OCL, which is proposed in [12]. Like the work in [23] our work aims to realize graph transformation in USE towards an OCL-based framework for model transformation. The core of our approach is the OCL, which is also the central ingredient in modeling and transformation languages.

VII. Conclusion and Future Work

This paper has introduced a tool as an extension of USE for presenting precisely models and transformations. This tool is a realization of our model-driven approach based on the integration of TGGs and OCL. We have illustrated our approach by explaining how this tool can fully support the example transformation between statecharts and extended hierarchical automata.

We have explained our OCL-based approach for operationalizing triple rules including OCL. Transformation operations derived from triple rules including OCL are realized by taking two views on them: Declarative OCL pre- and postconditions are employed as operation contracts, and USE command sequences are taken as an operational realization. Due to the full OCL support of USE, our tool provides an on-the-fly verification of model transformations and transformation quality assurance such as checking well-formedness of models and maintaining model consistency.

Our future work includes the following issues. We will accomplish the SC2EHA transformation with further triple rules. We plan to enhance features of our tool, including the user interface of our tool for manipulating transformation rules, and algorithms for applying them.

We focus on extending the USE4TGG language, especially, for its expressiveness. This will be carried out in various case studies. Studying the role of invariants in describing Domain-Specific Languages (DSLs) and the transformation of DSLs into core modeling languages is also our aim in future.

REFERENCES

[1] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, 1st ed. Wiley, Aug. 2004.

[2] 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.

[3] 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.

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

[5] 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.

[6] R. Wagner, "Developing Model Transformations with Fujaba," in *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, H. Giese and B. Westfechtel, Eds., vol. tr-ri-06-275. University of Paderborn, 2006.

[7] D. Varró and A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics)," *Software and Systems Modeling*, vol. 2, no. 3, pp. 187–210, Oct. 2003.

[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] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, "MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations," in *Model Driven Architecture – Foundations and Applications*, A. Rensink and J. Warmer, Eds., vol. 4066. Springer Berlin, 2006, pp. 361–375.

[10] J. de Lara and H. Vangheluwe, "AToM3: A Tool for Multi-formalism and Meta-modelling," in *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2002, pp. 174–188.

[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] 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.

[13] D. Varró, M. Asztalos, D. Bisztray, A. Boronat, D.-H. Dang, R. Geiß, J. Greenyer, P. V. Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell, "Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools," in *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited* , ser. LNCS, A. Schürr, M. Nagl, and A. Zündorf, Eds., vol. 5088. Springer, 2008, pp. 540–565.

[14] D.-H. Dang, "Triple Graph Grammars and OCL for Validating System Behavior," in *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, ser. LNCS, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds., vol. 5214. Springer, 2008, pp. 481–483.

[15] 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.

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

[17] 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.

[18] F. Büttner and M. Gogolla, "Realizing Graph Transformations by Pre- and Postconditions and Command Sequences," in *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds. Springer Berlin, LNCS 4178, 2006, pp. 398–413.

[19] D. Varró, "A Formal Semantics of UML Statecharts by Model Transition Systems," in *Proceedings of the First International Conference on Graph Transformation*, vol. 2505. Springer-Verlag, 2002, pp. 378–392.

[20] D. Latella, I. Majzik, and M. Massink, "Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, Dec. 1999.

[21] M. Gogolla, "(An Example for) Metamodeling Syntax and Semantics of Two Languages, their Transformation, and a Correctness Criterion," in *Proc. Dagstuhl Seminar on Language Engineering for Model-Driven Software Development*, J. Bezivin and R. Heckel, Eds. http://www.dagstuhl.de/04101/, 2004.

[22] 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.

[23] M. Gogolla, F. Büttner, and D.-H. Dang, "From Graph Transformation to OCL using USE," in *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited*, ser. LNCS, A. Schürr, M. Nagl, and A. Zündorf, Eds., vol. 5088. Springer, 2008, pp. 585–586.