Model Transformation using Constraint Handling Rules as a basis for Model Interpretation

Marcel Dausend and Frank Raiser

Institute of Software Engineering and Compiler Construction
Ulm University, Ulm, Germany

Abstract. In this paper, we present a model transformation approach aiming to simplify automatic processing of UML state machine models, especially for interpretation. The main requirements are easing the implementation of the interpreter and reducing the number of calculations necessary to execute a model.
Our model transformation preserves the semantics and is implemented using CHR. The result of the transformation is an UML state machine model based on the concept of compound transitions. Furthermore we provide an interpreter for those models which supports a comprehensive subset of UML state machine concepts, i. a. junction, fork, join.
Our preliminary results show that state machine interpreters can profit from the former model transformation. It simplifies certain aspects of the interpreter implementation and positively affects the performance of the interpreter, e.g. regarding transition selection and transition execution.

Keywords: Constraint Handling Rules (CHR), Unified Modeling Language (UML), model transformation, model interpretati!on

1 Introduction

The Unified Modeling Language (UML)[1] is the de facto standard in Software Engineering for modeling software systems. An important and popular application of UML models is automatic processing like static evaluation, or interpretation, and also code generation. Interpretation of UML state machines poses a hard problem for several reasons: regarding numerous static as well as dynamic aspects, handling complex models, and dealing with non formal UML semantics, etc.

Optimization of UML state machine interpreters, programs that execute a given model by interpretation, is a challenging topic, especially enhancing the performance without violating UML semantics or excluding certain concepts. We think, there are at least two effective strategies for optimization: enhancing the implementation, and/or simplifying the model before interpretation.

In this paper, we illustrate how the interpretation can profit from model simplification by former model transformation (cf. Sec. 4). Our strategy of model

transformation is to consolidate information spread over the model advantageously at specific model elements. The basis for this transformation is the UML concept of compound transitions, i.e. a whole transition path between sets of states contained in a state machine model. We implemented a model transformation using Constraint Handling Rules (CHR) to make compound transitions explicit by folding multiple transitions. Thereby, information needed for interpretation is no longer spread over the model. In consequence, the interpretation (cf. Sec. 5) can profit by a simplified transition selection algorithm as well as transition execution.

We implemented the model transformation using CHR, a general-purpose programming language, formally defined as a state transition system that is both rule- and logic-based (cf. Sec. 3). CHR displays the same tendency as many rule- or logic-based approaches in that there already exist hundreds of scientific publications on it, yet we are only just beginning to witness its adoption by industry users.

2 State Machines

In this section, we introduce the basic concepts of UML state machines. For further reading on this subject (cf. [1–3]).

State machines are a language package of UML used to model behavior. Each behavior model is based on a context provided by a classifier that defines properties, operations, visibility, etc. A state machine can define its own context or can be executed in an existing context if it is invoked by a different behavior within that context.

The static structure of UML is defined by the UML meta-model given by means of class diagrams and a textual specification for each element following the structure: Generalizations, Description, Attributes, Associations, Constraints, Semantics, and Notation. Figure 1 gives an overview of a part of the UML state machine meta-model. All these elements and their related constraints and semantics (cf. [1, Ch. 15]) are relevant for both our model transformation and execution of the resulting models by interpretation.

The two main concepts of the meta-model (cf. Fig. 1) are classes to define elements, and associations to define the relations between classes. The class StateMachine is associated with the class Region by a composition. This means that a state machine must have at least one region that only exists as long as its state machine. Each region contains some vertexes and transitions between them. The UML distinguishes between different kinds of vertexes using inheritance. A Vertex of type State can be simple, composite, or orthogonal. A so-called Pseudostate is used as intermediate targets of a transition path. Each kind of pseudo state (cf. PseudostateKind in Fig. 1) has its own semantics taking into account individual aspects of the dynamics of state changes. Behavioral aspects are realized by associating Behavior, Trigger, and Constraint elements with a State and/or Transitions.

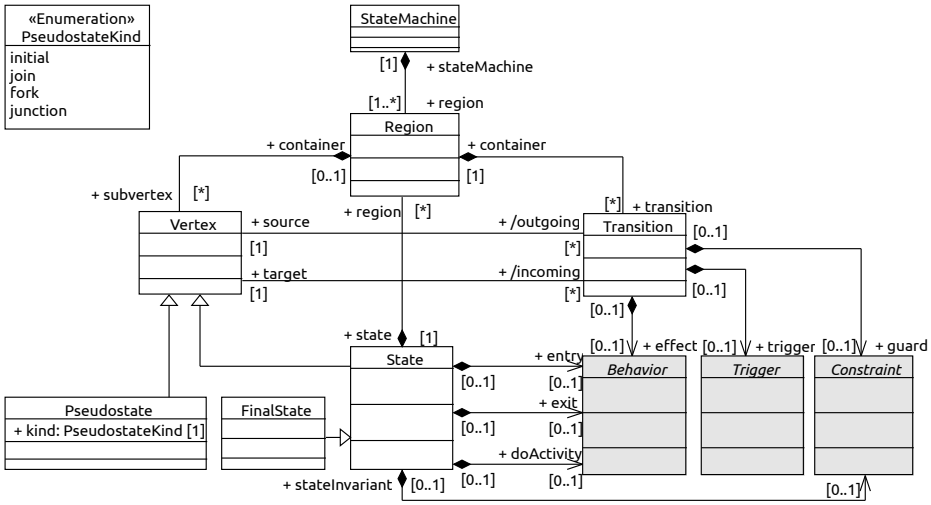

Fig. 1. Excerpt of the UML state machine meta-model showing all model elements for which we support model transformation and interpretation. (Behavior, Trigger and Constraint are classes from different UML meta-model packages.)

The dynamic aspects of state machines are given by a textual description [1]. The configuration of a state machine is represented by a set of active states called active state configuration.

"Junction vertices are semantic-free vertices that are used to chain together multiple transitions" [1, P. 557]. They allow branching of transitions into several paths where a path is selected at runtime based on the active state configuration.

Figure 2 shows a state machine containing one region *rtop* with an initial, a fork *fork1*, and a join pseudo state *join1*, and two states *a* and *b*. The transitions *t3* up to *t7* are connected to either *fork1* or *join1* pseudo state and form a compound transition. A compound transition leads from a set of states to another set of states. The states *a* and *b* are orthogonal states, which means in terms of UML that these states have at least two regions. State *a* has two regions: *r1* containing state *a1*, and region *r2* with an initial state and a sub-state *a2*. State *b* owns three regions *r3*, *r4*, and *r5*. Each region contains a state which is the target of one transition *t6*, *t7*, or *t8*. Region *r5* contains an initial pseudo state. which is the source of *t8*.

3 Constraint Handling Rules

Constraint Handling Rules has been introduced in the early 90ies [4]. The article [4] summarized its early development and was the main reference for the following decade, but was recently replaced by a new book [5].

CHR distinguishes user-defined or CHR constraints from built-in constraints. The latter are provided by a host system, e.g. , Prolog or Java. A CHR program

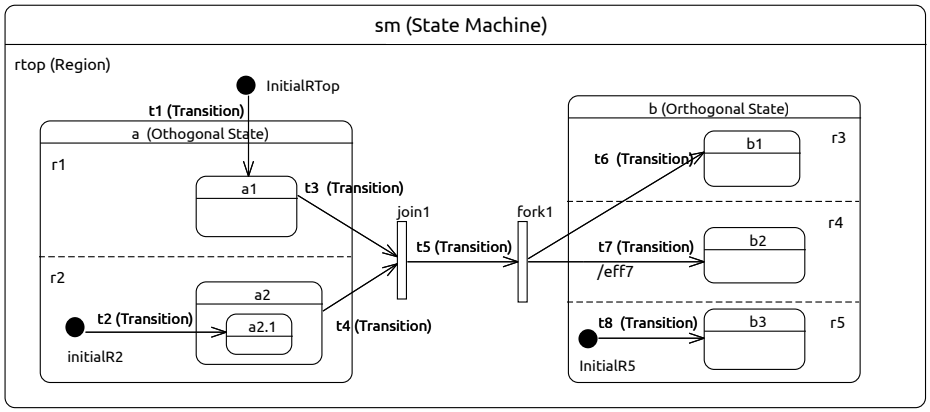

Fig. 2. A UML state machine containing two orthogonal states a and b.

consists of a finite set of rules, for which the remainder of this section discusses their syntax and semantics.

Definition 1 (CHR Rules [6]).

A CHR program $\mathcal{P}$ *is a finite set of* rules *of the form* $H_1 \backslash H_2 \Leftrightarrow G \mid B$, *where* H_1 *and* H_2 *– called the* head *– are multisets of CHR constraints,* G *– called the* guard *– is a conjunction of built-ins, and* $B = B_c, B_b$ *– called the* body *– consists of a multiset* B_c *of CHR constraints and a conjunction* B_b *of built-ins.* H_1 *is referred to as the* kept head *and* H_2 *as the* removed head.

There are different specializations of CHR rules, depending on the head:

- Simpagation rules *(*$H_1 \neq \emptyset \wedge H_2 \neq \emptyset$*) written as* $H_1 \backslash H_2 \Leftrightarrow G \mid B$
- Simplification rules *(*$H_1 = \emptyset \wedge H_2 \neq \emptyset$*) written as* $H_2 \Leftrightarrow G \mid B$
- Propagation rules *(*$H_1 \neq \emptyset \wedge H_2 = \emptyset$*) written as* $H_1 \Longrightarrow G \mid B$

Every CHR rule can optionally be preceded by an identifier *(or* name*) followed by* @*. Finally, if the guard* G *is* $\top$ *it may be omitted together with the* $\mid$ *character.*

Variables that occur in the rule, but not in its head, are called local variables.

There exist multiple operational semantics of CHR rules in the literature [7]. A concise formal definition for the equivalence-based operational semantics [8] is given below. It is suitable for our semantics-related discussions in this work, while our implementation relies on the refined semantics [9] instead. The refined semantics removes multiple sources of non-determinism from the equivalence-based operational semantics, and hence, forms the basis of many available CHR implementations.

Definition 2 (Operational Semantics ω_e).

For a CHR program $\mathcal{P}$*, the state transition system* $(\Sigma_e/\equiv_e, \rightarrowtail_e)$ *is given in Table 1. The transition is based on a variant of a rule* r *in* $\mathcal{P}$ *such that its local*

variables are disjoint from the variables occurring in the representation of the pre-transition state.

$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{[\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle] \rightarrowtail_e^r [\langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle]}$$

Table 1. State Transition System ω_e with Equivalence Classes

In contrast to other definitions, a CHR state in ω_e is an equivalence class that contains all possible syntactic formulations. This results in the concise definition of the state transition system given in Table 1. As can be seen from this inference rule, a rule application removes the H_2 part of a head and adds the body to the current state if the guard G is satisfied.

In this work, we interpret UML diagrams as typed graphs. Earlier work [6] discussed how graphs, more precisely graph transformation systems, can be embedded in CHR. The mapping itself is fairly intuitive and encodes nodes and edges as CHR constraints. Additionally, to achieve proper semantics for the graph transformation systems, the degree of nodes is encoded explicitly in the corresponding CHR constraints and consistently updated by rules. Due to the intuitive nature of the encoding, we omit a detailed discussion here, and instead refer the interested reader to [6].

4 Transformation into Compound State Machine Models

In this section we introduce the supported concepts (cf. Sec. 4.1), explain and demonstrate our model transformation by means of our example (cf. Sec.4.2), and characterize the resulting models (cf. Sec. 4.3).

> In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define [a] path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). [1, P. 589]

Our model transformation basically creates a single compound transition for every transition path of a state machine model – we call the resulting model a *compound state machine* model. Such a model offers considerable advantages for further processing of state machines – in our case interpretation.

4.1 Supported UML Concepts

Our transformation concept preserves all information of the source model such that the resulting model is capable to substitute the original model. In the following, we refer to the most relevant UML concepts taken into account for our model transformation (cf. Fig. 1).

A simple compound transition comprises a source and a target State connected via a path of two Transitions with an intermediate Junction within the same region. To support this kind of compound transitions exit- and entry-Behavior of States, Trigger, Guard, and Effect of a Transition, and Junction are considered for model transformation.

Composite States contain one Region, e.g. with other States. Transitions from (or to) such an inner State (cf. state *a2.1* in Fig. 2) cross the border of at least one composite state so that their exit- and entry-Behavior are respected as well.

Orthogonal States extend the concept of compound transitions by allowing parallel Regions within a State (cf. states *a* and *b* in Fig. 2). Join and Fork enable modeling of explicit transitions between sets of states. The source or target of a compound transition is a set of States (cf. Fig. 2).

Further UML concepts are realized by our interpreter based on models resulting from our model transformation. Apart from explicit transitions, implicit ones, e.g. a default transition of a region (cf. Fig. 2), are handled by our interpreter. Furthermore, concepts like ConnectionPoints and Histories are not affected by our model transformation and therefore can be considered as well. Decisions are not taken into account because their semantics influence the transition selection by allowing side effects on guards of outgoing transitions during traversing.

4.2 Model Transformation with CHR

We illustrate the transformation by means of the example shown in Fig. 2. The transformation consists of several steps: preprocessing of the UML source model and its conversion into CHR code, model transformations called "transition folding" and "transition lifting", and post-processing. The main steps of the general model transformation are folding transition paths, lifting of transitions, and generating code.

The UML source model is assumed to be valid according to the UML specification [1]. The preprocessing ensures that suitable termination conditions can be found for recursive transformation rules. Therefore, every incoming transition of Junctions, if more than one exists, is assigned to its own new Junction. The outgoing transitions of the new Junctions are copies of those of the original Junction.

The resulting UML model is considered as CHR input as basis for model transformation. According to the Fork and its related elements in Fig. 2 a formal description, given as 'XML Metadata Interchange' file (cf. Lst. 1.1), is converted into equivalent CHR code (cf. Lst. 1.2).

Preprocessing rules We defined nine CHR constraints according to the meta-model of UML state machines [1]. These CHR constraints reflect the UML elements of state machines and the relation between them, e.g. a state as a source or target of a transition (cf. Lsts. 1.1, 1.2). Nesting relations between elements are made explicit introducing an `owner` constraint, e.g. `owner(`*`fork1`*`,` *`rtop`*`)` means that *fork1* is owned by region *rtop*. The ownership of Transitions is

not determined by the UML specification. Therefore, we consider a TRANSITION to be owned by the least common ancestor REGION of its source and target STATES. For example transition *t2* is owned by region *r2* whereas *t4* is owned by *rtop* (cf. Fig. 2). As last preprocessing step, all else guards of transitions are made explicit by setting them to the conjunction of all negated guards of the transitions originating from the same source vertex. This step eases further processing of guards during transformation as well as interpretation.

```
...
<region xmi:id="4" name="rtop">
...
  <subvertex xmi:type="uml:Pseudostate" xmi:id="16"
      name="Fork" kind="fork1"/>
  <subvertex xmi:type="uml:State" xmi:id="17" name="b">
    <region xmi:id="22" name="r3">
      <subvertex xmi:type="uml:State" xmi:id="23" name="b1"/>
    </region>
    <region xmi:id="24" name="r4">
      <subvertex xmi:type="uml:State" xmi:id="25" name="b2"/>
    </region> ...
  </subvertex>
  <transition xmi:id="27" name="t5" source="10" target="16"/>
  <transition xmi:id="28" name="t6" source="16" target="23"/>
  <transition xmi:id="29" name="t7" source="16" target="25"
      effect="eff7"/>
</region>
...
```

Listing 1.1. XMI Definition for the relevant model elements of our example.

```
...
region(rtop, 0, 11), owner(rtop, sm),
vertex(b1, 2), owner(b1, r3), vertex(b2, 2), owner(b2, r4),
vertex(fork1, 4), pstate(fork1, fork), owner(fork1, rtop),
...
trans(t5, [join1], [fork1], [join1], [fork1], [], [], []),
    src(t5, join1), tgt(t5, fork1), owner(t5, rtop),
trans(t6, [fork1], [b1], [fork1], [b1], [], [], []),
    src(t6, fork1), tgt(t6, b1), owner(t6, rtop),
trans(t7, [fork1], [b2], [fork1], [b2], [], [], [eff7]),
    src(t7, fork1), tgt(t7, b2), owner(t7, rtop)
...
```

Listing 1.2. Part of the CHR input model relevant for folding the transitions connected to the fork.

Transition folding. Folding means merging two consecutive TRANSITIONS of a transition path which are coupled via either a FORK, JOIN or JUNCTION. For each of these pseudo states two CHR rules for folding are defined. One for the base case and one for the recursion case. The interesting transformation part in

our example (cf. Fig. 2) deals with the transition path between the states *a1*, *a2*, *b1*, and *b2* involving PSEUDOSTATES *join1* and *fork1* as well as TRANSITIONS *t3* to *t7*. All these TRANSITIONS are recursively transformed into a single semantics preserving compound transition.

Regarding FORKS, the aim of each recursive transformation step is to remove one outgoing transition and shift its information to a different outgoing transition. If only one outgoing transition remains, the base case removes the fork and both transitions and adds its information to a new transition. In Fig. 2 three transitions (*t5*-*t7*) are connected to the fork *fork1*.

The CHR constraints (before transformation) are shown in Lst. 1.2. Each CHR constraint has arguments to reflect the information of the UML model and additional information. The constraint `region` has three arguments: a name, depth, and degree. The `argument` degree is a counter for related elements, e.g. vertexes and transitions. Region *rtop* is the topmost region of the state machine and has a depth of zero by definition. At the beginning of the transformation, region *rtop* is related to 11 elements, i.e. 11 other constraints make use of rtop. The constraint `vertex` has two arguments, name and degree. The `pstate` constraint indicates the kind of pseudo state, i.e. *fork1* is of type FORK. `Transition` has eight arguments: the name of the transition, a list of states to deactivate, a list of states to activate, a list of source states, a list of target states, a list of triggers, a list of guards, and a list of effects. Thus, it holds all information of a compound transition.

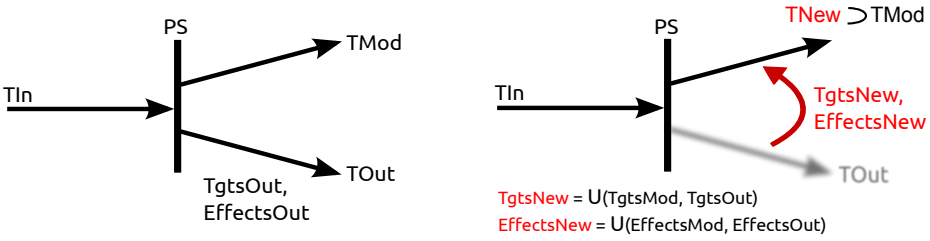

Fig. 3. Folding transformation applied on transitions connected to a fork pseudo state with at least two outgoing transitions (recursion case).

The head of the CHR rule in Lst. 1.3 describes the situation before the folding transformation. This situation is present in Lst. 1.2 and is shown schematically in Fig. 3 (left). The schematic arguments are mapped to the example as follows: *TIn* to *t5*, *PS* to *fork1*, *TMod* to *t6*, and *TOut* to *t7*. *TOut* targets vertex *V2* which is mapped to *b2*. All transitions are owned by region *rtop*. Because it is not determined by the rule which transition has to be chosen for removal, *TOut* (*t7*) is selected non-deterministically for removal.

The application of the rule removes the constraints for two outgoing transitions (*TMod* and *TOut*) of a pseudo state *PS*. These transitions are removed by applying the rule whereas a new one (*TNew*) is constructed to substitute *TMod*.

```
tgt(TIn, PS), pstate(PS, fork),
src(TMod, PS), tgt(TMod, _),
\
trans(TOut, _, _, _, TgtsOut, _, _, EffectsOut), src(TOut,
    PS), tgt(TOut, V2),
trans(TMod, _, ActsMod, SrcsMod, TgtsMod, TriggersMod,
    GuardsMod, EffectsMod),
vertex(V2, DV2),
vertex(PS, DPS)
<=>
append(EffectsOut, EffectsMod, EffectsNew),
append(TgtsMod, TgtsOut, TgtsNew),
trans(TNew, _, ActsMod, SrcsMod, TgtsNew, TriggersMod,
    GuardsMod, EffectsNew),
make_lca_owner(TIn, TMod, TNew), % compute and assign owner of TNew
remove_owner(TMod), remove_owner(TOut), % decrease degree of owners
TNew = TMod,
DV21 is DV2-1, vertex(V2, DV21),   % decrease degree of V2 and PS
DPS1 is DPS-1, vertex(PS, DPS1).
```

Listing 1.3. CHR rule of the recursive case to fold a Fork.

In order to preserve all information of both removed transitions, all arguments of *TMod* are assigned to *TNew*. Since UML only allows Effects on outgoing transitions, the effects of *TOut* and *TMod* are merged to a new list of effects for *TNew*. The lists of target vertexes are treated similarly. Finally, the degrees of the affected vertexes and regions (which own the transitions) are adjusted. As a consequence of removing *TOut* (*t7*), the degrees of *PS* (*fork1*), *V2* (*b2*), and the owning regions (*rtop*) have changed. Therefore, the `vertex` constraints are in the removed head and are recreated in the body with a degree which is decreased by one. The degree of the topmost region is decreased by two and increased by one by the `remove_owner` and `make_lca_owner` constraints (cf. Lst. 1.3). The resulting set of constraints is given in Lst. 1.4.

```
...
region(rtop, 0, 10), owner(rtop, sm),
vertex(b1, 2), owner(b1, r3), vertex(b2, 1), owner(b2, r4),
vertex(fork1, 3), pstate(fork1, fork), owner(fork1, rtop),
...
trans(t5, [join1], [fork1], [join1], [fork1], [], [], []),
    src(t5, join1), tgt(t5, fork1), owner(t5, rtop),
trans(t6, [fork1], [b1, b2], [fork1], [b1, b2], [], [],
    [eff7]), src(t6, fork1), tgt(t6, b1), owner(t6, rtop),
...
```

Listing 1.4. The CHR constraints after a transformation where Lst. 1.3 is applied on Lst. 1.2 (t6 $\sim$ *TMod* and *t7* $\sim$ *TOut*)

For the base case of the above transformation, the fork *PS* has exactly one incoming and one outgoing transition. Then, both transitions are removed and

a new transition is inserted as substitution. Unlike the recursive case the list of states to be deactivated, triggers, and guards from the incoming transition are taken into account.

Transition lifting. Another important concept of our model transformation is lifting of source and target transition ends. The lifting process is the last step before a transition is transformed into code. If a transition execution is performed leading to a state outside the region of its source state, all states whose borders are crossed have to be either deactivated or activated. The lifting transformation collects this information and stores it in the corresponding arguments of the constraint `trans`. A transition that cannot be lifted any more is transformed into code.

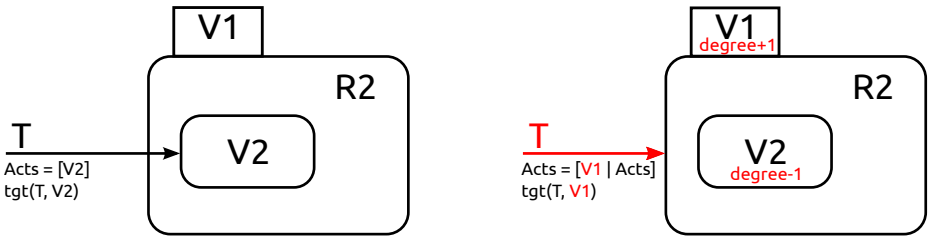

Fig. 4. Transformation to lift a target end of a transition to its surrounding state. This transformation collects the information about which states have to be activated during the execution and writes it to the modified transition.

```
region(R2, _, _), owner(R2, V1), owner(V2, R2)
\
vertex(V1, DV1), vertex(V2, DV2),
trans(T, Deacts, Acts, Srcs, Tgts, Trigger, Guard, Effect),
tgt(T, V2)
<=>
tgt(T, V1), % redefine target for transition T
trans(T, Deacts, [V1|Acts], Srcs, Tgts, Trigger, Guard,
  Effect), % append V1 to list of Acts
DV11 is DV1+1, vertex(V1, DV11), % increase degree of V1
DV21 is DV2-1, vertex(V2, DV21). % decrease degree of V2
```

Listing 1.5. The CHR rule lifts the target of a transition T from a vertex V1 to a vertex V2 where V2 owns the region R2 which contains V1.

Figure 4 illustrates the lifting of a transition's target end (left to right). If a vertex *V2* is a target of a transition *T*, and it is owned by a region *R2* then the transition end is "lifted" to the region's owner vertex *V1*. This condition is expressed by the kept head of the CHR rule in Lst. 1.5. The degree of those vertexes, the list of to be activated vertexes (*Acts*), and the target for the transition *T* are changed by the rule. Lifting results in increasing the degree of *V1* by one and in decreasing the degree of *V2* by one. In the case of execution of *T*, *V1* has to be activated before entering *V2*. Therefore, *V1* is appended to list *Acts* of *T*. Finally, the constraint `tgt` for *T* is changed from `tgt(T, V2)` to `tgt(T, V1)`.

Lifting terminates, if no parent vertex for *V2* exists. A analogous lifting is performed on source transition ends to construct the list *Deacts* of states to be deactivated.

Post-processing As a consequence, both lists – *Deacts* and *Acts* – contain an identical prefix of undesired vertexes. These vertexes are common ancestor states of the source and target of the transition, i.e. the transition connects vertexes inside a common parent vertex. Post-processing rules are added to eliminate these prefixes from the lists *Deacts* and *Acts* of a `transition`. As mentioned above, further rules deal with other aspects of the compound transition UML model (cf. Sec. 4.1), e.g. transition paths involving JUNCTIONS.

At last, all sub-models which are atomic in terms of our model transformation are translated into code constraints reflecting the final UML model. This model is the basis for our interpreter.

4.3 Model Transformation Summary

The resulting model according to Sec. 4.2 can be characterized as follows: The model consists of only STATES, REGIONS, TRANSITIONS, and elements which are unaffected by the transformation. Except for TRANSITIONS originating from INITIAL PSEUDO STATES, every TRANSITION directly connects two sets of STATES. Each TRANSITION holds all information necessary for its execution. This information is no longer distributed over the model. All "else"-GUARDS are now explicit, such that they can be analyzed in a context-free manner. Paths, that constitute a non regular compound transition (i.e. unsatisfiable GUARDS, or multiple TRIGGERS) are removed from the model.

5 UML Interpreter for Compound State Machines

The interpretation of UML models in general is a challenging topic. The complexity of UML itself and its imprecise semantics are very hard problems for automatic processing of UML models. Unclarities and ambiguities according to the semantics are resolved by following the semantics defined by [10, 11]. Currently we do not know any interpreter for models of UML state machines based on the concept of compound transitions. For that reason we implemented a prototypical interpreter as proof of concept.

In the following we outline the technical basis and concepts of the interpreter. Thereby, we focus on similarities and differences to traditional implementation concepts [12] and exemplarily refer to the state machine semantics of [10].

The interpreter is implemented in the Scala programing language[1]. Additionally, the actors package is used to support multi threading and ease communication between objects, e.g. states and regions. The interpreter comprises the Scala classes Application, StateMachine, State, Region, Transition, and Event. The class Application implements a given UML model according to our transformation result. Each other class realizes the behavior of the UML concepts their

[1] http://www.scala-lang.org/

name refer to. All these classes are implemented as agents to support parallelism and simplify communication.

First, we enumerate some concepts of the interpreter which are similar to traditional UML interpreter implementations. The interpreter is implemented in an object-oriented manner where behavioral objects correspond to UML objects. Some superordinate activities of the UML model execution are done by the state machine, e.g. event selection and event dispatching.

From our point of view, the main differences compare to other interpreters can be found in transition selection and transition execution. The UML requires so-called run-to-completion processing [1, p. 580]. Therefore, two points have to be guaranteed:

1. "an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.", i.e. a compound transition may only have one trigger.
2. "an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation."

As consequence, one has to assure that if a state configuration is left another stable state configuration will be reached. To guarantee this, it is necessary to check if all guards of a transition path will evaluate to true and that no more than one trigger is contained on the whole path (cf. [10]). This calculation costs a lot of resources at runtime.

For the transition selection based on our transformed model, where every transition already is a compound transition, both number of triggers and guard conditions can directly be checked without needing to traverse the model.

A state configuration change is controlled by the selected transition. The state change is organized in three phases: Deactivation of the transition's source states, execution of its effects, and activation of its target states.

For instance, executing the compound transition in Fig. 2 means, that state **a** and all its sub-states are exited, then the transition's effects are executed, and at last **b** and its sub-states are entered.

By introducing a propagation concept, the implementation executes transitions in compliance to the ordering of actions as specified in the UML specification [1]. The interpreter was tested on several different of our transformed models. So far, the implementation is limited to those concepts mentioned in Sec. 4.1.

6 Discussion and Conclusion

In this paper, we have presented a concept for model transformation of state machines aiming information consolidation by folding and lifting transformations applied on compound transitions of UML state machines (cf. Sec. 2). The resulting models are used for model interpretation. In conclusion, the implementation of the interpreter profits by simplified transition selection algorithm as well as transition execution.

Refactoring of UML state machines by graph transformation was already performed by [13]. They use model transformation to normalize diagrams for better understanding of the semantics in terms of the notation itself. In the contrary, our model transformation focuses on transformations that simplify automatic processing of state machines, especially for interpretation.

Folli and Mens [14] introduce model transformations to refactor class and state machine diagrams by AGG as proof of concept for graph transformation for model refactoring.

We chose a pragmatic approach for both, definition and interpretation of our model transformation using CHR (cf. Sec. 4) and implementation of the interpreter (cf. Sec. 5). Our approach provides precise and straightforward definitions of models and model transformations.

As basis for a short comparison to related model transformation approaches, we briefly classify our transformation according to the feature model of [15]. We confine our application to unidirectional model-to-model transformation. We use CHR which naturally supports the following model transformation features: variables, patterns, logic expressions, i.e. both, executable as well as non-executable logic, separation of left-hand side from right-hand side of a rule, in-place transformation, and conditional rule selection. Furthermore, scheduling, iteration and phasing are supported implicitly. Certainly, CHR supports no scoping, i.e. all rules are checked for an entire CHR input.

Several current model transformation approaches tend to QVT [16], the model transformation standard of the OMG [17, 18]. These approaches and other ones provide powerful (diagrammatic) specification of transformation systems based on user definable meta-models [19, 20, 14, 21]. These approaches are notably beneficial for large scaled model driven development approaches. Certainly, those techniques, e.g. bidirectional transformation approaches according to QVT, maybe more complicated also if not needed as in our case. Unfortunately, the current QVT Mapping Language is far less intuitive and easy-to-use in case of unidirectional transformations [18]. [22] compare QVT and graph transformation by transforming UML state machines into Communication Sequential Processes specifications. They conclude that both approaches are rather similar. Graph transformation has its power in the clear operational idea which enhances rule specification whereas the above mentioned approaches benefit from the bidirectionality idea. According to [23] graph transformation can support bi-directionality just as well.

Others transformation languages are domain specific. For example XSLT[2], a declarative XML-based language for model transformation of XML documents seems to be suitable for our approach, too. Agreeing with [15], we think that the scalability of XSLT has limitations regarding manual implementation of model transformation. Handwritten XSLT based implementations are hard to maintain because verbosity and readability issues of XMI and XSLT. CHR provides a good

[2] XSTL (eXtensible Stylesheet Language Transformation) is a XML technology used for describing (mainly syntactic) transformations between XML files.

readability, clear semantics and additionally offers a good basis for profitability of certain concepts of an implementation.

Several approaches using model transformation successfully apply Prolog, CHR and/or CLP for different purposes, e.g. as transformation engine [24, 19, 25]. They agree that logic programming and constraint programming are valuable for model transformation.

Pretschner and Lotzbeyer [24] state that CLP and CHR are well suited for interactive generation techniques based on symbolic model execution as demonstrated in [26], too. VIATRA [19] uses Prolog for the declaration of the transformation including control flow graphs and graph transformation rules. The author states that automated transformation from source to target model using Prolog, which is hidden from the user, provides a more efficient solution as by using XSLT.

Our interpreter is implemented in Scala and is based on the formally defined UML semantics of [11]. The interpreter was implemented as proof of concept to test our resulting models and to analyze the benefits for the implementation itself. In our opinion the transition selection as well as the transition execution algorithms of our implementation profit by localization of model information resulting in a better performance of our interpretation.

7 Future Work

We are currently extending our model transformation approach using CHR, and the preliminary results are encouraging. Based on these results, the most relevant topics to substantiate, apply and extend our current work are:

- Proving of the semantics preserving property for each CHR rule
- Analyzing our approach to achieve qualitative as well as quantitative results about its benefits and drawbacks
- Applying our approach on other, similar UML diagrams, e.g. Activity Diagrams

Aside these topics, our approach can be applied to simplify code generation or test case generation approaches.

References

1. OMG. Unified Modeling Language Superstructure v2.3, 2010.
2. R. Miles and K. Hamilton. *Learning UML 2.0.* O'Reilly Media, Inc., 2006.
3. B. Selic. The Theory and Practice of Modelling Language Design for Model-Based Software Engineering — A Personal Perspective. *Lecture Notes in Computer Science*, 6491:290–321, 2011.
4. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37:95–138, 1998.
5. T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.
6. F. Raiser. *Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis.* PhD thesis, Ulm University, Germany, 2010.

7. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As Time Goes By: Constraint Handling Rules – A Survey of CHR Research between 1998 and 2007. *Theory and Practice of Logic Programming*, 10(1):1–47, 2010.
8. T. Frühwirth and F. Raiser, editors. *Constraint Handling Rules – Compilation, Execution, and Analysis.* Books on Demand GmbH, Norderstedt, 2011.
9. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, September 2004.
10. M. Dausend. Entwicklung einer ASM-Spezifikation der Semantik der Zustandsautomaten der UML 2.0. Master's thesis, Universität Ulm, 2007.
11. J. Kohlmeyer. *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2.* PhD thesis, Universität Ulm, 2009.
12. A. Kirshin, D. Dotan, and A. Hartman. A UML Simulator Based On a Generic Model Execution Engine. *Lecture Notes in Computer Science*, 4364:324, 2007.
13. M. Gogolla and F. Parisi-Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In *Workshop on Precise Semantics for Modelling Techniques*, volume TUM-I9803, pages 55–72. TU München, 1998.
14. A. Folli and T. Mens. Refactoring of UML models using AGG. *ECEASST*, 8, 2007.
15. K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, pages 1–17, Anaheim, CA, USA, 2003.
16. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation. 1.1, 2011.
17. A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, 2005.
18. A. Balogh and D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1280–1287. ACM, 2006.
19. D. Varró. Automated Program Generation for and by Model Transformation Systems. *Applied Graph Transformation (AGT'02)*, pages 161–174, 2002.
20. L. Geiger and A. Zündorf. Statechart Modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 127(1):37–49, 2005.
21. B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *Software Language Engineering*, pages 227–244, 2009.
22. J.M. Küster, S. Sendall, and M. Wahler. Comparing two Model Transformation Approaches. In *Workshop on OCL and Model Driven Engineering*, 2004.
23. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Logical constraints for managing non-determinism in bidirectional model transformations. In *Model-Driven Engineering, Logic and Optimization: friends or foes? (MELO 2011)*, 2011.
24. A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01)*, volume 2, Toronto, 2001.
25. J.M. Almendros-Jiménez and L. Iribarne. ODM-based UML Model Transformations using Prolog. In *Model-Driven Engineering, Logic and Optimization: friends or foes? (MELO 2011)*, 2011.
26. A. Ciarlini and T. Frühwirth. Automatic Derivation of Meaningful Experiments for Hybrid Systems. In *Proc. ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, 2000.