Electronic Communications of the EASST
Volume 39 (2011)

Graph Computation Models
Selected Revised Papers from the
Third International Workshop on
Graph Computation Models (GCM 2010)

A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation

Ulrike Golas, Enrico Biermann, Hartmut Ehrig and Claudia Ermel

24 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation

Ulrike Golas[1], Enrico Biermann[2], Hartmut Ehrig[2] and Claudia Ermel[2]

[1] golas@zib.de Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany
[2] enrico|ehrig|lieske@cs.tu-berlin.de Technische Universität Berlin, Germany

Abstract: Several different approaches to define the formal operational semantics of statecharts have been proposed in the literature, including visual techniques based on graph transformation. These visual approaches either define a compiler semantics (translating a concrete statechart into a semantical domain) or they define an interpreter using complex control and helper structures. Existing visual semantics definitions make it difficult to apply the classical theory of graph transformations to analyze behavioral statechart properties due to the complex control structures.

In this paper, we define an interpreter semantics for statecharts based on amalgamated graph transformation where rule schemes are used to handle an arbitrary number of transitions in orthogonal states in parallel. We build on an extension of the existing theory of amalgamation from binary to multi-amalgamation including nested application conditions to control rule applications for automatic simulation. This is essential for the interpreter semantics of statecharts. The theory of amalgamation allows us to show termination of the interpreter semantics of well-behaved statecharts, and especially for our running example, a producer-consumer system.

Keywords: operational semantics, statecharts, graph transformation, amalgamation

1 Introduction and Related Work

In [Har87], Harel introduced statecharts by enhancing finite automata by hierarchies, concurrency, and some communication issues. Over time, many versions with slightly differing features and semantics have evolved. In the UML specification [OMG09], the semantics of UML state machines is given as a textual description accompanying the syntax, but it is ambiguous and explained essentially by examples. In [Bee02], a structured operational semantics (SOS) for UML statecharts is given based on the preceding definition of a textual syntax for statecharts. The semantics combines Kripke structures and an auxiliary semantics using deduction such that a semantical step is a transition step in the Kripke structure. This semantics is difficult to understand due to its non-visual nature. The same problem arises in [RACH00], where labeled transition systems and algebraic specification techniques are used.

There are also different approaches to define a visual rule-based semantics of statecharts. One of the first was [MP96], where for each transition t a transition production p_t is derived describing the effects of the corresponding transition step. A similar approach is followed in [Kus01], where first a state hierarchy is constructed explicitly, and then a semantical step is given by a complex transformation unit constructed from the transition rules of a maximum set of independently enabled transitions. In [KGKK02], in addition, class and object diagrams are integrated. The approach highly depends on concrete statechart models and is not a general interpreter semantics for statecharts. Moreover, problems arise for nesting hierarchies, because the resulting situation is not fixed but also depends on other current or inactive states. In [GP98], the hierarchies of statecharts are flattened to a low-level graph representing an automaton defining the intended semantics of the statechart model. This is an indirect definition of the semantics, and the resulting transformation rules are model-specific and not applicable to statecharts in general.

In [Var02], Varró defines a general interpreter semantics for statecharts. His intention is to separate syntactical and static semantic concepts (like conflicts, priorities etc.) of statecharts from their dynamic operational semantics, which is specified by graph transformation rules. To this end, he uses so-called model transition systems to control the application of the operational rules, which highly depend on additional helper structures encoding activation or conflicts of transitions and states.

Amalgamation is important for graph transformations in order to model synchronized parallelism of rules with shared subrules and corresponding transformations. For example, it has been applied to applications of parallel graph transformations to communication-based systems [TB94] and to model transformations from BPMN to BPEL [BEE+10]. The concept of amalgamation was first developed for the synchronization of two rules [BFH87] and then extended to that of an arbitrary number of rules [Tae96] and integrated in the well-known theory of $\mathscr{M}$-adhesive systems [GEH10]. Using amalgamation for the definition of an operational semantics, the main advantage of our solution is that we do not need helper structures or a complex external control structure to cover the complex statecharts semantics: we define a state transition mainly by one interaction scheme followed by some clean-up rules. Therefore, our model-independent definition based on rule amalgamation is not only visual and intuitive but allows us to show termination and forms a solid basis for applying further graph transformation-based analysis techniques.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to our model of statecharts as typed attributed graphs. In Section 3, we review the basic ideas of algebraic graph transformation [EEPT06] and give a short introduction to amalgamated transformation based on [GEH10], which is used for the operational semantics of statecharts in Section 4. Based on the given semantics, we discuss the formal analysis of termination of semantical steps in statecharts. The operational semantics is demonstrated along a sample statechart modeling a producer-consumer system in Section 5. In Section 6, the implementation in our tool Henshin is presented. Finally, Section 7 concludes our paper and considers future work directions.

2 Modeling of Statecharts

In this section, we model statecharts by typed attributed graphs. We restrict ourselves to the most interesting parts of the statechart diagrams: we allow orthogonal regions as well as state nesting. But we do not handle entry and exit actions on states, nor extended state variables, and we allow guards only to be conditions over active states.

In Figure 1, the sample statechart ProdCons is depicted modeling a producer-consumer system. When initialized, the system is in the state prod, which has three regions. There, in parallel a producer, a buffer, and a consumer may act. Parallel substates are modelled in orthogonal regions of a common superstate (separated by dashed lines), which means that while the superstate is active, also exactly one substate from each orthogonal region is active. The producer alternates between the states produced and prepare, where the transition produce models the actual production activity. It is guarded by a condition that the parallel state empty is also current, meaning that the buffer is empty and may receive a product, which is then modeled by the action incbuff denoted after the /-dash. Similarly to the producer, the buffer alternates between the states empty and full, and the consumer between wait and consumed. The transition consume is again guarded by the state full and followed by a decbuff-action emptying the buffer. Two possible events may happen causing a state transition to leave the state prod: the consumer may decide to finish the complete run; or there may be a failure detected after the production leading to the error-state. After repair, the error-state can be exited via the corresponding exit-transition and the standard behavior in the prod-state is executed again.

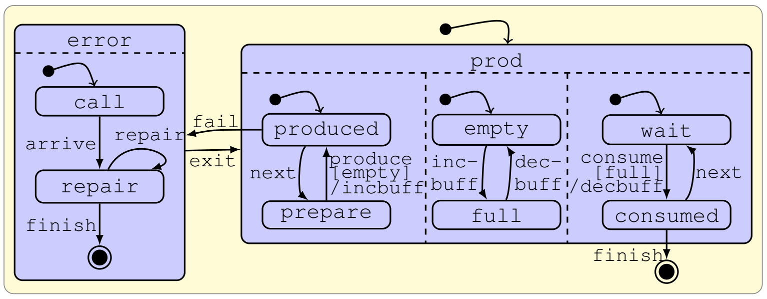

Figure 1: Sample statechart ProdCons

For our statechart language, we use typed attributed graphs, which are an extension of typed graphs by attributes [EEPT06]. We do not give details here, but use an intuitive approach, where the attributes of a node are given in a class diagram-like style. For the values of attributes in the rules we can also use variables.

The type graph TG_{SC} is given in Figure 2. We use multiplicities to denote some constraints directly in the type graph. To obtain valid statechart models, additional constraints are defined in Figure 3. We use nested conditions, which are defined explicitely in Section 3 and can be intuitively understood as the requirement to find occurrences of the morphism's domain and codomain in the target object leading to commuting diagrams. Note that i_A defines the unique morphism from an initial object I to some object A. Each diagram consists of exactly one statemachine SM (constraint c_1) containing one or more orthogonal regions R. A region contains states S, where state names are unique within one region. A state may again contain one or more regions. Constraint c_2 expresses in addition that each region is contained in either exactly one state or the statemachine. Moreover, states may be initial (attribute value isInitial = true) or final (attribute value isFinal=true), each region has to contain exactly one initial and at most one final state, and

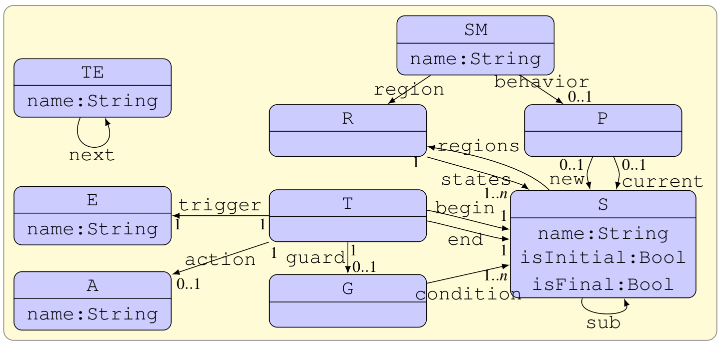

Figure 2: Type graph TG_{SC} for statecharts

final states cannot contain regions (constraint c_3). Note that the edge type `sub` is only necessary to compute all substates of a state, which we need for the definition of the semantics. This relation is computed in the beginning using the `states`- and `regions`-edges.

A transition `T` begins and ends at a state, is triggered by an event `E`, and may be restricted by a guard `G` and followed by an action `A`. A guard has one or more states as conditions. There is a special event with attribute value `name="exit"` which is reserved for exiting a state after the completion of all its orthogonal regions, which cannot have a guard condition (constraint c_4). Moreover, final states cannot be the beginning of a transition and their name attribute has to be set to `name="final"` (constraint c_5). In addition, transitions cannot link states in different orthogonal regions (constraint c_6), which means that both regions are directly contained in the same state. A pointer `P` describes the active states of the statemachine. Note that newly inserted current states are marked by a `new`-edge, while for established current states the `current`-edge is used (which is assumed to be the standard type and thus not marked in our diagrams). This differentiation is necessary for the semantics, where we need to distinguish between states that were current before and states that just became current in the last state transition. Trigger elements `TE` describe the events which have to be handled by the statemachine. Note that they do not necessarily form a queue because orthogonal states may lead to parallel triggers which are sequentialized by the semantics. For simplicity we still call it event queue. There are at least the empty trigger element with attribute value `name = null` and no outgoing `next`-edge, and exactly one pointer in each diagram (constraint c_7). The pointer and trigger elements are used later for the description of the operational semantics, but they do not belong to the general syntactical description.

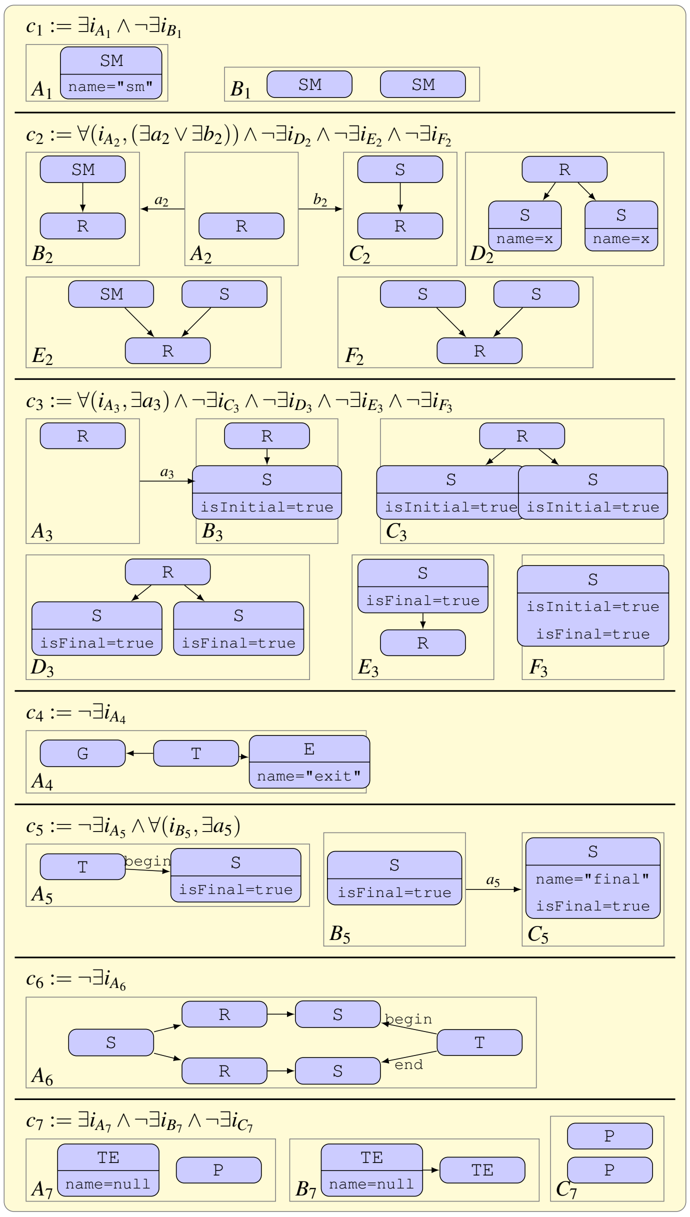
$c_1 := \exists i_{A_1} \wedge \neg\exists i_{B_1}$
SM name="sm" A_1
SM SM B_1
$c_2 := \forall(i_{A_2}, (\exists a_2 \vee \exists b_2)) \wedge \neg\exists i_{D_2} \wedge \neg\exists i_{E_2} \wedge \neg\exists i_{F_2}$
SM R B_2
a_2
R A_2
b_2
S R C_2
R S name=x S name=x D_2
SM S R E_2
S S R F_2
$c_3 := \forall(i_{A_3}, \exists a_3) \wedge \neg\exists i_{C_3} \wedge \neg\exists i_{D_3} \wedge \neg\exists i_{E_3} \wedge \neg\exists i_{F_3}$
R A_3
a_3
R S isInitial=true B_3
R S isInitial=true S isInitial=true C_3
R S isFinal=true S isFinal=true D_3
S isFinal=true R E_3
S isInitial=true isFinal=true F_3
$c_4 := \neg\exists i_{A_4}$
G T E name="exit" A_4
$c_5 := \neg\exists i_{A_5} \wedge \forall(i_{B_5}, \exists a_5)$
T begin S isFinal=true A_5
S isFinal=true B_5
a_5
S name="final" isFinal=true C_5
$c_6 := \neg\exists i_{A_6}$
S R S R S begin T end A_6
$c_7 := \exists i_{A_7} \wedge \neg\exists i_{B_7} \wedge \neg\exists i_{C_7}$
TE name=null P A_7
TE name=null TE B_7
P P C_7

Figure 3: Constraints limiting the valid statecharts

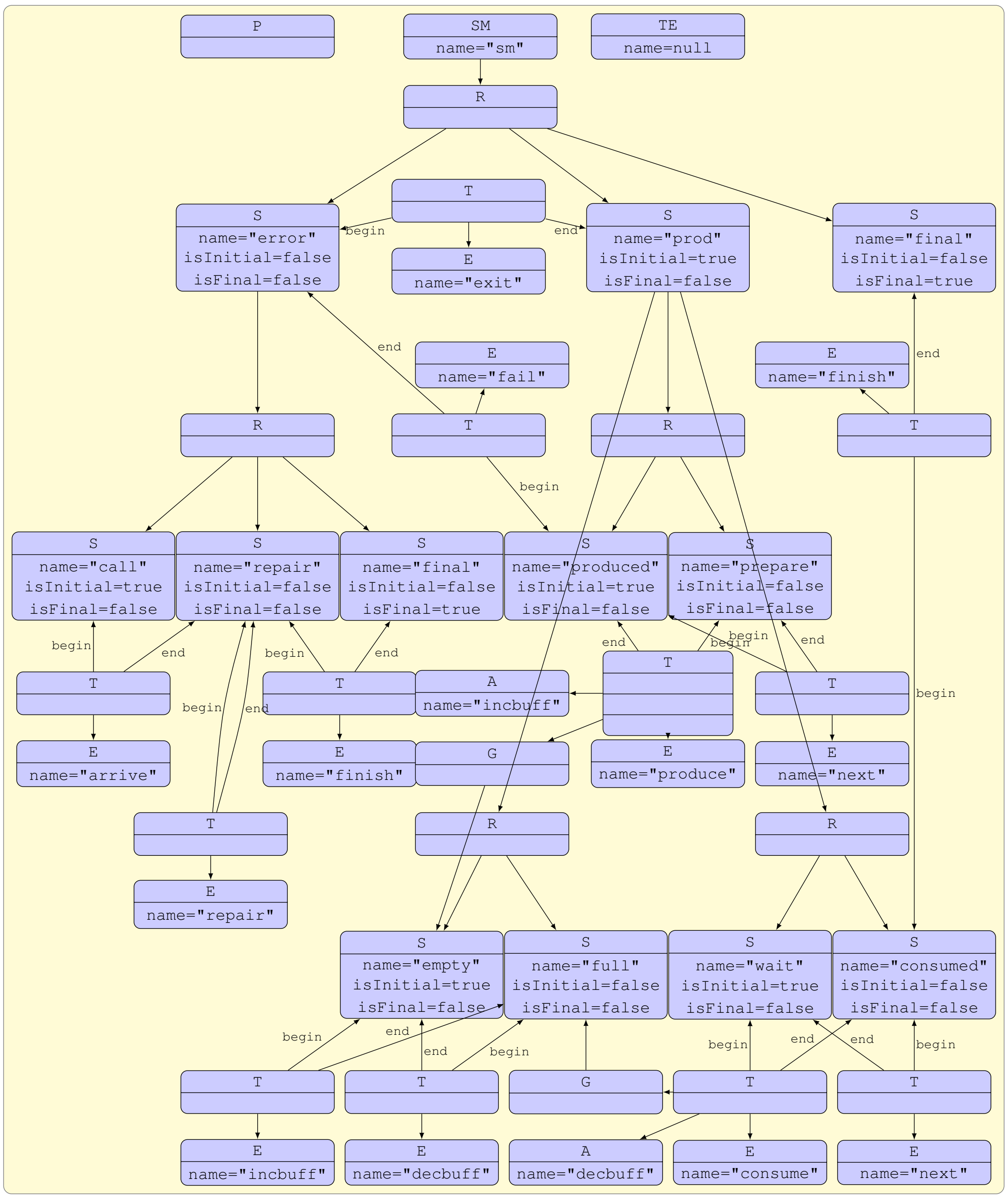
P
SM name="sm"
TE name=null
R
T
E name="exit"
S name="error" isInitial=false isFinal=false
S name="prod" isInitial=true isFinal=false
S name="final" isInitial=false isFinal=true
E name="fail"
E name="finish"
S name="call" isInitial=true isFinal=false
S name="repair" isInitial=false isFinal=false
S name="final" isInitial=false isFinal=true
S name="produced" isInitial=true isFinal=false
S name="prepare" isInitial=false isFinal=false
A name="incbuff"
G
E name="arrive"
E name="finish"
E name="produce"
E name="next"
E name="repair"
S name="empty" isInitial=true isFinal=false
S name="full" isInitial=false isFinal=false
S name="wait" isInitial=true isFinal=false
S name="consumed" isInitial=false isFinal=false
E name="incbuff"
E name="decbuff"
A name="decbuff"
E name="consume"
E name="next"
begin
end

Figure 4: Statechart `ProdCons` in abstract syntax

In Figure 4, the sample statechart `ProdCons` from Figure 1 is depicted in abstract syntax. Nodes `P` and `TE` are added, which have to exist for a valid statechart model but are not visible in the concrete syntax. For simulating statechart runs, the event queue of the statechart (consisting of only one default element named `null` in Figure 4) can be filled by events to be processed (see Figure 12 in Section 5 for a possible event queue for our sample statechart).

Since edges of types `sub`, `behavior`, `current`, and `next` only belong to the semantics but not to the syntax of statecharts, we leave them out for the definition of the language of statecharts. All attributed graphs typed over this reduced type graph $TG_{SC,Syn}$ satisfying all the constraints are valid statecharts.

Definition 1 (Language VL_{SC}) Define the syntax type graph $TG_{SC,Syn} = TG_{SC} \setminus \{\texttt{sub}, \texttt{behavior}, \texttt{current}, \texttt{next}\}$ based on the type graph TG_{SC} in Figure 2. The language VL_{SC} consists of all typed attributed graphs respecting the type graph $TG_{SC,Syn}$ and the constraints in Figure 3, i. e. $VL_{SC} = \{(G, type) \mid type : G \to TG_{SC,Syn}, G \models c_1 \wedge \ldots \wedge c_7\}$.

3 Introduction to Amalgamated Graph Transformation

In this section, we review the basic ideas of algebraic graph transformation [EEPT06] and give a short introduction into amalgamated transformation based on [GEH10], to be used for the interpreter semantics of statecharts in Section 4.

A graph grammar $GG = (RS, SG)$ consists of a set of rules RS and a start graph SG. A rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$ consists of a left-hand side L, an interface K, a right-hand side R, two injective graph morphisms $L \xleftarrow{l} K$ and $K \xrightarrow{r} R$, and an application condition ac on L. Applying a rule p to a graph G means to find a match m of L in G, given by a graph morphism $m : L \to G$ which satisfies the application condition ac, and to replace this matched part $m(L)$ by the corresponding right-hand side R of the rule. By $G \overset{p,m}{\Longrightarrow} H$, we denote the direct graph transformation where rule p is applied to G with match m leading to the result H. The formal construction of a direct transformation is a double-pushout (DPO) as shown in the diagram with pushouts (PO_1) and (PO_2) in the category of graphs. The graph D is the intermediate graph after removing $m(L)$, and H is constructed as gluing of D and R along K. A graph transformation is a sequence of direct transformations, denoted by $G \overset{*}{\Longrightarrow} H$, and the graph language $L(GG)$ of graph grammar GG is the set $L(GG) = \{G \mid SG \overset{*}{\Longrightarrow} G\}$ of all graphs derivable from SG.

$$\begin{tikzcd}
ac \arrow[r, phantom, "\triangleright"] & L \arrow[d, "m"'] \arrow[dr, phantom, "(PO_1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(PO_2)"] & R \arrow[d] \\
& G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

An important concept of algebraic graph transformation is parallel and sequential independence of graph transformation steps leading to the Local Church–Rosser and Parallelism Theorem [Roz97], where parallel independent steps $G \overset{p_1,m_1}{\Longrightarrow} G_1$ and $G \overset{p_2,m_2}{\Longrightarrow} G_2$ lead to a parallel transformation $G \overset{p_1+p_2,m}{\Longrightarrow} H$ based on a parallel rule $p_1 + p_2$. If p_1 and p_2 share a common subrule p_0, the amalgamation theorem in [BFH87] shows that a pair of "amalgamable" transformations $G \overset{(p_i,m_i)}{\Longrightarrow} G_i$ $(i = 1, 2)$ leads to an amalgamated transformation $G \overset{\tilde{p},\tilde{m}}{\Longrightarrow} H$ via the amalgamated rule $\tilde{p} = p_1 +_{p_0} p_2$ constructed as gluing of p_1 and p_2 along p_0. The concept of amalgamable transformations is a weak version of parallel independence, with independence outside the subrule match, and amalgamation can be considered as a kind of "synchronized parallelism".

For the interpreter semantics of statecharts we need an extension of amalgamation in [BFH87] w.r.t. three aspects: first, we need a family of rules $p_1, \ldots, p_n$ with a common subrule p_0 for $n \geq 2$; second, we need typed attributed graphs [EEPT06] instead of "plain graphs", and third, we need rules with application conditions.

In the following, we formulate the extended amalgamation concept for a general notion of graphs and application conditions, where *general graphs* are objects in a weak adhesive HLR category [EEPT06] and *general application conditions* are nested application conditions [HP09], including positive and negative ones and their combinations by logic operators. For readers not familiar with weak adhesive HLR categories and nested application conditions, it is sufficient to think of rules based on graphs and (typed) attributed graphs with positive and/or negative application conditions (see [EEPT06] for more details).

A match $m : L \to G$ satisfies a positive (negative) condition of the form $\exists a$ ($\neg\exists a$) for $a : L \to N$ if there is a (no) injective $q : N \to G$ with $q \circ a = m$. More general, $m : L \to G$ satisfies a nested condition of the form $\exists(a, ac_N)$ on L with condition ac_N on N if there is an injective $q : N \to G$ with $q \circ a = m$ and q satisfies ac_N. Note that $\forall(a, ac_N)$ is denoted as $\neg\exists(a, \neg ac_N)$ (see application conditions in Figure 9 and Figure 10).

An important concept is the shift of ac on L along a morphism $t : L \to L'$ s.t. for all $m' \circ t : L \to G$, m' satisfies $\mathit{Shift}(t, ac)$ if and only if $m = m' \circ t : L \to G$ satisfies ac [EHL10].

$$\begin{tikzcd}
ac \arrow[r, phantom, "\rhd"] & L \arrow[r, "t"] \arrow[dr, "m"'] & L' \arrow[d, "m'"] \arrow[r, phantom, "\lhd"] & \mathit{Shift}(t,ac) \\
 & & G &
\end{tikzcd}$$

Based on [GEH10], we are now able to introduce amalgamated rules and transformations with a common subrule p_0 of $p_1, \ldots, p_n$. A kernel morphism describes how the subrule is embedded into the larger rules.

Definition 2 (Kernel morphism). *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ for $i = 0, \ldots, n$, a* kernel morphism *$s_i : p_0 \to p_i$ consists of morphisms $s_{i,L} : L_0 \to L_i$, $s_{i,K} : K_0 \to K_i$, and $s_{i,R} : R_0 \to R_i$ such that in the diagram on the right (1_i) and (2_i) are pullbacks and (1_i) has a pushout complement for $s_{i,L} \circ l_0$, i.e. $s_{i,L}$ satisfies the gluing condition w.r.t. l_0. The pullbacks (1_i) and (2_i) mean that K_0 is the intersection of K_i with L_0 and also of K_i with R_0.*

$$\begin{tikzcd}
L_0 \arrow[d, "s_{i,L}"'] \arrow[dr, phantom, "(1_i)"] & K_0 \arrow[l, "l_0"'] \arrow[r, "r_0"] \arrow[d, "s_{i,K}"] \arrow[dr, phantom, "(2_i)"] & R_0 \arrow[d, "s_{i,R}"] \\
L_i & K_i \arrow[l] \arrow[r] & R_i
\end{tikzcd}$$

Definition 3 (Amalgamated rule and transformation). *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ for $i = 0, .., n$ with kernel morphisms $s_i : p_0 \to p_i$ ($i = 1, \ldots, n$), then the amalgamated rule $\tilde{p} = (\tilde{L} \longleftarrow \tilde{K} \longrightarrow \tilde{R}, \tilde{ac})$ of $p_1, \ldots, p_n$ via p_0 is constructed as the componentwise gluing of $p_1, \ldots, p_n$ along p_0, where $\tilde{ac}$ is the conjunction of $\mathit{Shift}(t_{i,L}, ac_i)$. $\tilde{L}$ is the gluing of $L_1, \ldots, L_n$ with shared L_0 leading to $t_{i,L} : L_i \to \tilde{L}$. Similar gluing constructions lead to $\tilde{K}$ and $\tilde{R}$ and we obtain kernel morphisms $t_i : p_i \to \tilde{p}$ and $t_i \circ s_i = t_0$ for $i = 1, \ldots, n$. We call p_0* kernel rule, *and $p_1, \ldots, p_n$* multi rules. *An* amalgamated transformation *$G \overset{\tilde{p}}{\Longrightarrow} H$ is a transformation via the amalgamated rule $\tilde{p}$.*

$$\begin{tikzcd}
p_0 \arrow[r, "t_0"] \arrow[dr, "s_i"'] & \tilde{p} \\
 & p_i \arrow[u, "t_i"']
\end{tikzcd}$$

Example 1 (Amalgamated rule construction) *We construct an amalgamated rule for the initialization of a statemachine with two orthogonal regions. A pointer has to be linked to the statemachine and to the initial states of both the statemachine's regions. Rules are depicted in a compact notation where we do not show the interface K. It can be inferred by the intersection $L \cap R$. The mappings are given as numberings for nodes and can be inferred for edges. The kernel rule p_0 in Figure 5 models the linking of the pointer to the statemachine. We have two multi-rules p_1 and p_2 modelling the linking of the pointer to the initial states of two different*

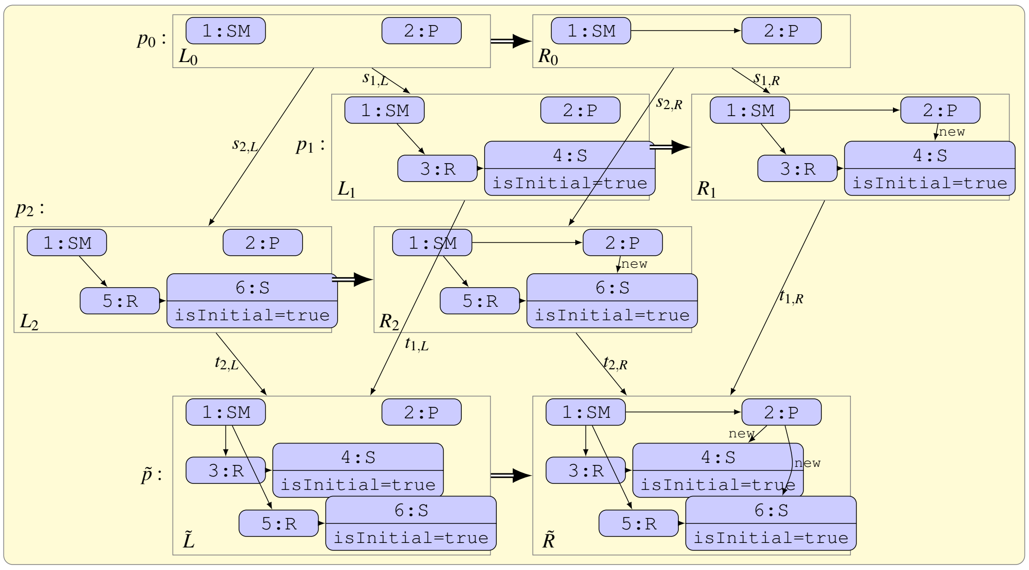

Figure 5: Construction of amalgamated rule

regions. In the amalgamated rule $\tilde{p}$, the common subaction (linking the pointer to the statemachine) is represented only once since the multi-rules p_1 and p_2 have been glued at the kernel rule p_0. The kernel morphisms are $t_i : p_i \to \tilde{p}$ for $i = 1,2$.

Given a bundle of direct transformations $G \overset{p_i,m_i}{\Longrightarrow} G_i$ $(i = 1,..,n)$, where p_0 is a subrule of p_i, we want to analyze whether the amalgamated rule $\tilde{p}$ is applicable to G combining all direct transformations. This is possible if they are *multi-amalgamable*, i.e. the matches agree on p_0 and are parallel independent outside. This concept of multi-amalgamability is a direct generalization of amalgamability in [BFH87] and leads to the following theorem [GEH10].

Theorem 1 (Multi-amalgamation) *Given rules $p_0,\dots,p_n$, where p_0 is a subrule of p_i, and multi-amalgamable direct transformations $G \overset{p_i,m_i}{\Longrightarrow} G_i$ $(i = 1,\dots,n)$, then there is an amalgamated transformation $G \overset{\tilde{p},\tilde{m}}{\Longrightarrow} H$.*

Proof Idea: Using the properties of the multi-amalgamable bundle, we can show that $\tilde{m}$ with $\tilde{m} \circ t_{i,L} = m_i$ induced by the colimit is a valid match for the amalgamated rule $\tilde{p}$ leading to the amalgamated transformation because the componentwise gluing is a colimit construction. For an extended proof idea see Thm. 2 in [GEH10], the complete proof can be found in [Gol11].

For many application areas, including the interpreter semantics of statecharts, we do not want to explicitly define the kernel morphisms between the kernel rule and the multi rules, but we want to obtain them dependent on the object to be transformed. In this case, only an interaction scheme $is = \{s_1,\dots,s_k\}$ with kernel morphisms $s_j : p_0 \to p_j$ $(j = 1,\dots,k)$ is given, which defines different bundles of kernel morphisms $s'_i : p_0 \to p'_i$ $(i = 1,\dots,n)$ where each p'_i corresponds to some p_j for $j \leq k$.

Definition 4 (Interaction scheme) A kernel rule p_0 and a set of multi rules $\{p_1, \dots, p_k\}$ with kernel morphisms $s_i : p_0 \to p_i$ form an interaction scheme $is = \{s_1, \dots, s_k\}$.

Given an interaction scheme, we want to apply as many rules p_j as often as possible over a certain match of the kernel rule p_0. In the following, we consider *maximal weakly disjoint matchings*, where we require the matchings of the multi rules not only to be multi-amalgamable, but also disjoint up to the match of the kernel rule, and maximal in the sense that no more valid matches for any multi rule in the interaction scheme can be found.

Definition 5 (Maximal weakly disjoint matching). *Given an interaction scheme $is = \{s_1, \dots, s_k\}$ and a tuple of matchings $m = (m_i : L'_i \to G)$ with $i = 1, \dots, n$, where each p'_i corresponds to some p_j for $j \le k$, with transformations $G \overset{p'_i, m_i}{\Longrightarrow} G_i$, then m forms a* maximal weakly disjoint matching *if the bundle $G \overset{p'_i, m_i}{\Longrightarrow} G_i$ is multi-amalgamable, the square $(P_{i\ell})$ is a pullback for all $i \neq \ell \in \{1, \dots, n\}$, and for any rule p_j no other match $m' : L_j \to G$ can be found such that $((m_i), m')$ fulfills this property.*

$$\begin{tikzcd}
L_0 \arrow[r, "s_{i,L}"] \arrow[d, "s_{\ell,L}"'] \arrow[dr, phantom, "(P_{i\ell})"] & L'_i \arrow[d, "m_i"] \\
L'_\ell \arrow[r, "m_\ell"'] & G
\end{tikzcd}$$

Note that different matches may use the same rule p_j. The pullback requirement already implies the existence of the morphisms to show that the matches are parallel independent outside the kernel match. Only the property for the application conditions has to be checked in addition.

Proposition 1 *Given an object G, a bundle of kernel morphisms $s = (s_1, \dots, s_n)$, and matches $m_1, \dots, m_n$ leading to a bundle of direct transformations $G \overset{p_i, m_i}{\Longrightarrow} G_i$ such that $m_i \circ s_{i,L} = m_0$ and square $(P_{i\ell})$ is a pullback for all $i \neq \ell$ then the bundle $G \overset{p_i, m_i}{\Longrightarrow} G_i$ is s-amalgamable for transformations without application conditions.*

Proof. By construction, the matches m_i agree on the match m_0 of the kernel rule. It remains to be shown that they are parallel independent outside the kernel match.

Given the transformations $G \overset{p_i, m_i}{\Longrightarrow} G_i$ with pushouts (20_i) and (21_i), consider the following cube, where the bottom face is pushout (20_i), the back right face is a pullback by definition, and the front right face is pullback (P_{ij}). Now construct the pullback of f_i and m_j as the front left face, and from $m_j \circ s_{j,L} \circ l_0 = m_i \circ s_{i,L} \circ l_0 = m_i \circ l_i \circ s_{i,K} = f_i \circ k_i \circ s_{i,K}$ we obtain a morphism p with $\hat{f} \circ p = s_{j,L} \circ l_0$ and $\hat{m} \circ p = k_i \circ s_{i,K}$.

$$\begin{tikzcd}
L_i \arrow[d, "m_i"'] \arrow[dr, phantom, "(20_i)"] & K_i \arrow[l, "l_i"'] \arrow[r, "r_i"] \arrow[d, "k_i"] \arrow[dr, phantom, "(21_i)"] & R_i \arrow[d, "n_i"] \\
G & D_i \arrow[l, "f_i"] \arrow[r, "g_i"'] & G_i
\end{tikzcd}$$

$$\begin{tikzcd}
 & & K_0 \arrow[dll, "p"'] \arrow[drr, "l_0"] \arrow[dd, "s_{i,K}"] & & \\
P \arrow[dr, "\hat{f}"] \arrow[dd, "\hat{m}"'] & & & & L_0 \arrow[dlll, "s_{j,L}"'] \arrow[dd, "s_{i,L}"] \\
 & L_j \arrow[dd, "m_j"] & K_i \arrow[dll, "k_i"'] \arrow[drr, "l_i"] & & \\
D_i \arrow[dr, "f_i"'] & & & & L_i \arrow[dlll, "m_i"] \\
 & G & & &
\end{tikzcd}$$

From pullback composition and decomposition it follows that also the back left face is a pullback. Now the $\mathscr{M}$-van Kampen property leads to a pushout in the top face. Since pushout complements are unique up to isomorphism, P is isomorphic to K_j. Thus the morphism $p_{ji} := \hat{m}$ leads to parallel independence outside the kernel match. This construction can be applied for all pairs i, j leading to weakly parallel independent matches without application conditions. □

With this characterization of maximal weakly independent matches we obtain the following algorithm for their computation.

Algorithm 1 (Maximal weakly disjoint matching). *Given an object G and an interaction scheme $is = \{s_1, \dots, s_k\}$, a maximal weakly disjoint matching $m = (m_0, m_1, \dots, m_n)$ can be computed as follows:*

1. *Set $i = 0$. Choose a kernel matching $m_0 : L_0 \to G$ such that $G \xRightarrow{p_0, m_0} G_0$ is a valid transformation.*

2. *As long as possible: Increase i, choose a multi rule $\hat{p}_i = p_j$ with $j \in \{1, \dots, k\}$, and find a match $m_i : L_j \to G$ such that $m_i \circ s_{j,L} = m_0$, $G \xRightarrow{p_j, m_i} G_i$ is a valid transformation, $m_i \neq m_\ell$, the square $(P_{i\ell})$ is a pullback, and $\hat{p}_\ell$ is applicable to G_i via the extension of m_ℓ to G_i for all $\ell = 1, \dots, i-1$, i.e. the application condition $\hat{ac}_\ell$ is satisfied for this extended match.*

3. *If no more valid matches for any rule in the interaction scheme can be found, return $m = (m_0, m_1, \dots, m_n)$.*

Note, that we may find different maximal weakly disjoint matchings for a given interaction scheme, which may even lead to the same bundle of kernel morphisms. For a fixed maximal weakly disjoint match we can apply Theorem 1 leading to an amalgamated transformation $G \xRightarrow{\tilde{p}', \tilde{m}} H$, where $\tilde{p}'$ is the amalgamated rule of $p'_1, \dots, p'_n$ via p_0.

Given a set IS of interaction schemes is and a start graph SG, we obtain an amalgamated graph grammar with amalgamated transformations via maximal matchings, defined by maximal weakly disjoint matchings of the corresponding multi rules.

Definition 6 (Amalgamated graph grammar) An *amalgamated graph grammar* $AGG = (IS, SG)$ consists of a set IS of interaction schemes and a start graph SG. The *language* $L(AGG)$ of AGG is defined by $L(AGG) = \{G \mid \exists$ amalgamated transformation $SG \overset{*}{\Rightarrow} G$ via maximal matchings$\}$.

4 An Interpreter Semantics for Statecharts

The semantics of statecharts is modeled by amalgamated transformations, where one step in the semantics is modeled by several applications of interaction schemes. The main part of a state transition can be modeled by a single interaction scheme, but some additional rules are necessary to remove and add the proper pointers from and to hierarchical states. For the application of an interaction scheme we use maximal weakly disjoint matchings.

The termination of the interpreter semantics of a statechart in general depends on the structural properties of the simulated statechart. A simulation will terminate for the trivial cases that the event queue is empty, that no transition triggers an action, or that there is no transition from any active state triggered by the current head elements of the event queue. Since transitions may trigger actions which are added as new events to the queue it is possible that the simulation of a statechart may not terminate even if all semantical steps do. Hence, it is useful to define structural constraints that provide a sufficient condition guaranteeing termination of the simulation in general for well-behaved statecharts, where we forbid cycles in the dependencies of actions and events.

Definition 7 (Well-behaved statecharts) For a given statechart model, the *action-event graph* has as nodes all event names and an edge (n_1, n_2) if an event with name n_1 triggers an action named n_2.

A statechart is called *well-behaved* if it is finite, has an acyclic state hierarchy, and its action-event graph is acyclic.

Example 2 *An example of a well-behaved statechart is our statechart model in Figure 1. It is finite, has an acyclic state hierarchy, and its action-event graph is shown in Figure 6. This graph is acyclic, since the only action-event dependencies in our statechart occur between* `produce` *triggering* `incbuff` *and* `consume` *triggering* `decbuff`.

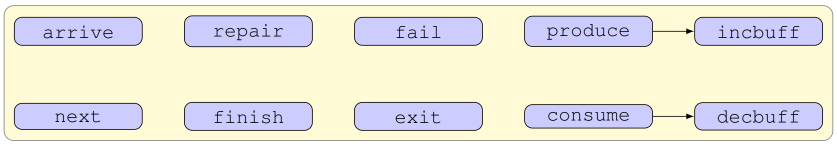

Figure 6: The action-event graph of our statechart example

The semantics of our statecharts is modeled by amalgamated transformations, but we apply the rules in a more restricted way, meaning that one step in the semantics is modeled by several applications of interaction schemes. We assume to have a finite statechart with a finite event queue where all trigger elements are already given in the diagram as an initial event queue.

For the *initialization step*, we compute all substates of all states by applying the rules `setSub` and `transSub` in Figure 7 as long as possible. Then, the interaction scheme `init` is applied followed by the interaction scheme `enterRegions` applied as long as possible, which are depicted in Figure 8. With `init`, the pointer is associated to the statemachine and all initial states of the statemachine's regions. The interaction scheme `enterRegions` handles the nesting and sets the current pointer also to the initial states contained in an active state. When applied as long as possible, this means that all substates are handled. Note that not all initial substates become

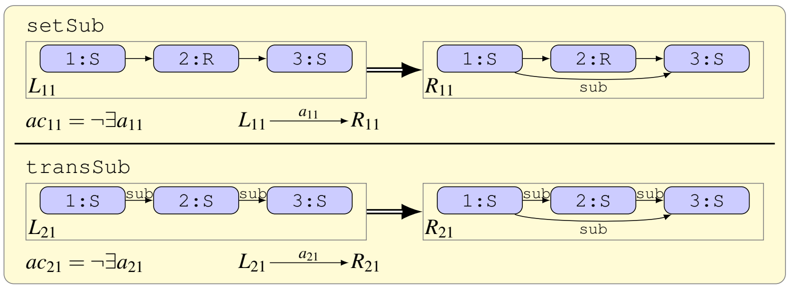
setSub
1:S
2:R
3:S
L_{11}
1:S
2:R
3:S
sub
R_{11}
$ac_{11} = \neg\exists a_{11}$
$L_{11} \xrightarrow{a_{11}} R_{11}$
transSub
1:S
sub
2:S
sub
3:S
L_{21}
1:S
sub
2:S
sub
3:S
sub
R_{21}
$ac_{21} = \neg\exists a_{21}$
$L_{21} \xrightarrow{a_{21}} R_{21}$

Figure 7: The rules `setSub` and `transSub`

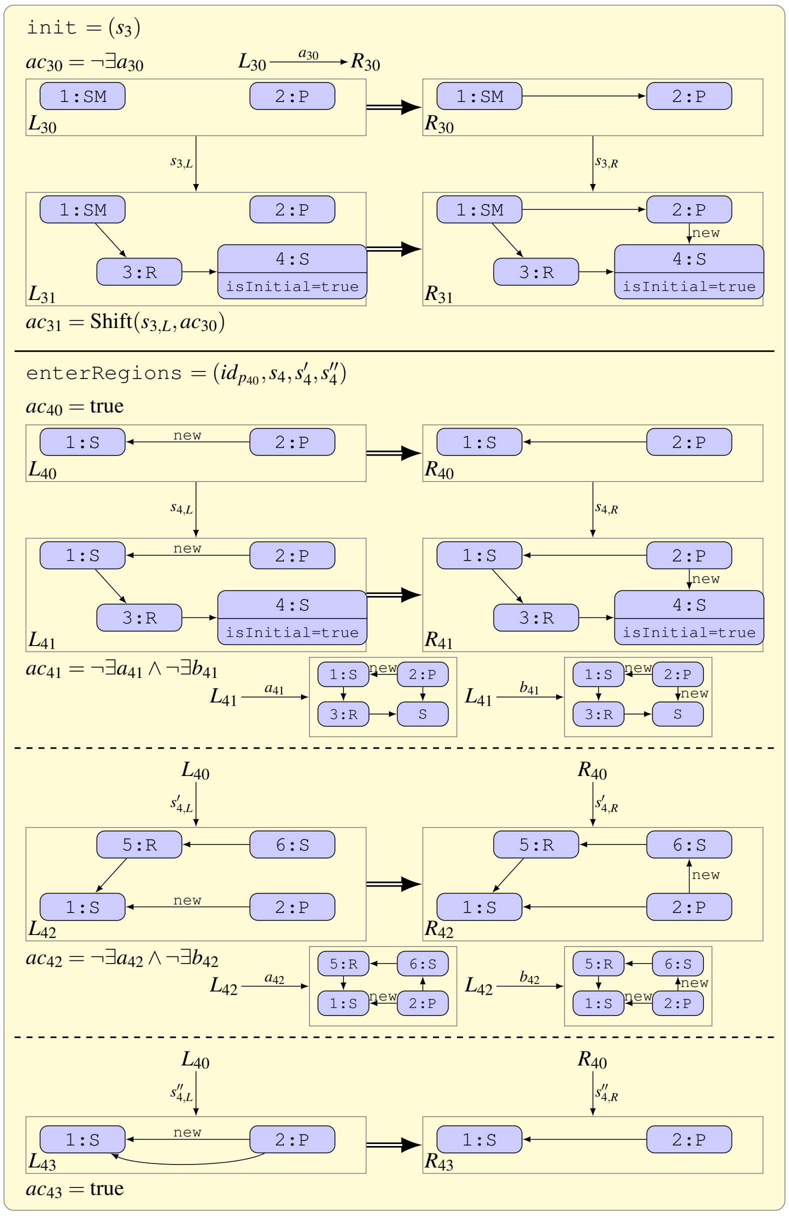

Figure 8: The interaction schemes `init` and `enterRegions`

active, but only those which are contained in a hierarchy of nested initial states. The interaction scheme `enterRegions` also contains the identical kernel morphism $id_{p_{40}} : p_{40} \to p_{40}$. Using maximal weakly disjoint matchings, an identical kernel morphism has the effect that the kernel rule can be applied without a valid match for any multi rule. This ensures that this kernel rule is also applied in the lowest hierarchy level changing the `new`- to a `current`-edge. For later use, also double edges are deleted and if the direct superstate is not marked by the pointer a `new`-edge is added to it.

The application of the rules setSub and transSub terminates because there could be at most one sub-edge between each pair of states due to the application conditions. Since no new states are created, these rules can only be applied finitely often.

The initialization step (applying init once and enterRegions as long as possible) terminates because the application of the interaction scheme enterRegions terminates: each application of enterRegions replaces one new-edge with a current-edge. The multi rules p_{41} and p_{42} create new new-edges on the next lower and upper levels of a hierarchical state, but if the state hierarchy is acyclic this interaction scheme is only applicable a finite number of times. The same holds for the multi rule p_{43} which deletes double edges, since the number of current- and new-edges is decreased. Thus, the transformation terminates.

Proposition 2 (Termination of initialization step) *For well-behaved statecharts, the initialization step terminates.*

A state transition representing a *semantical step*, i. e. switching from one state to another, is done by the application of the interaction scheme transitionStep shown in Figure 9 followed by the interaction schemes enterRegions!, leaveState1!, leaveState2!, and leaveRegions! given in Figure 8, 10, and 11 in this order, where ! means that the corresponding interaction scheme is applied as long as possible.

For such a semantical step, the first trigger element (or one of the first if more than one action of different orthogonal substates may occur next) is chosen and deleted, while the corresponding state transitions are executed. The application condition ac_{50} ensures that exit-trigger elements are handled with priority, because the rule is only applicable if for any existing exit-trigger element ($\forall a_{50}$) this is not a start element in the queue, i.e. it has a predecessor ($\exists b_{50}$). Moreover, it ensures that the chosen trigger element is a starting one, i.e. has no predecessor ($\neg\exists c_{50}$). Note that a transition triggered by its trigger element is active if the state it begins at is active, its guard condition state is active, and it has no active substate where a transition triggered by the same event is active. These restrictions are handled by the application conditions ac_{51} and ac_{52}. Moreover, if an action is provoked, this has to be added as one of the first next trigger elements. The two multi rules of transitionStep handle the state transition with and without action, respectively. The application condition ac_{52} is not shown explicitly, but the morphisms $a_{52}, \ldots, f_{52}$ are similar to $a_{51}, \ldots, f_{51}$ except that all objects contain in addition the node 8 : A.

The interaction schemes leaveState1, leaveState2, and leaveRegions handle the correct selection of the active states. When for a yet active state with regions, by state transitions all states in one of its regions are no longer active, also this superstate is no longer active, which is described by leaveState1. The interaction scheme leaveState2 handles the case that, when a state become inactive by a state transition, also all its substates become inactive. If for a state with orthogonal regions the final state in each region is reached then these final states become inactive, and if the superstate has an exit-transition it is added as the next trigger element. This is handled by leaveRegions.

For the termination of a semantical step it is sufficient to show that the four interaction schemes enterRegions, leaveState1, leaveState2, and leaveRegions are only applicable a finite number of times. For the interaction scheme enterRegions we have already argued that above. The interaction schemes leaveState1, leaveState2 as well as the multi rule

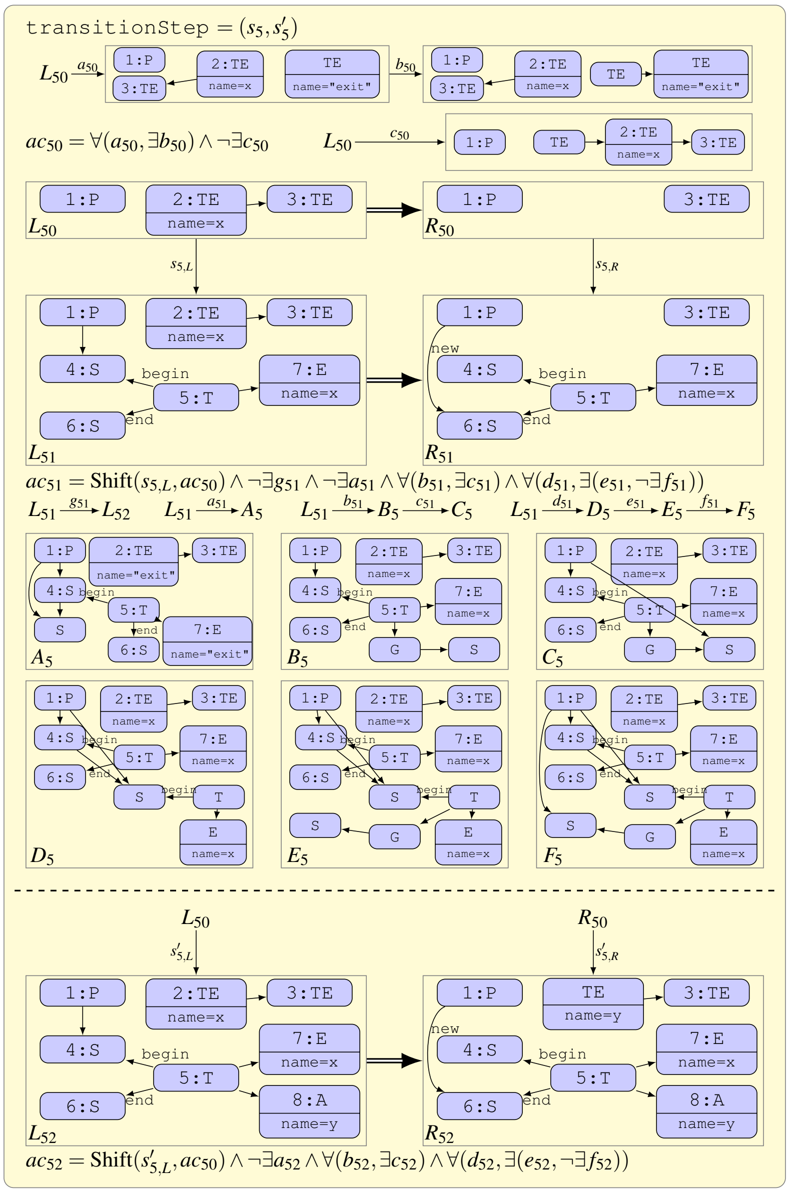
transitionStep = (s5, s'5)
ac50 = ∀(a50, ∃b50) ∧ ¬∃c50
ac51 = Shift(s5,L, ac50) ∧ ¬∃g51 ∧ ¬∃a51 ∧ ∀(b51, ∃c51) ∧ ∀(d51, ∃(e51, ¬∃f51))
ac52 = Shift(s'5,L, ac50) ∧ ¬∃a52 ∧ ∀(b52, ∃c52) ∧ ∀(d52, ∃(e52, ¬∃f52))
L50
R50
L51
R51
L52
R52
A5
B5
C5
D5
E5
F5
1:P
2:TE
3:TE
4:S
5:T
6:S
7:E
8:A
name=x
name=y
name="exit"
begin
end
new

Figure 9: The interaction scheme `transitionStep`

p_{81} of the interaction scheme `leaveRegions` reduce the number of active states in the statechart by deleting at least one `current`-edge. The application of the second multi rule p_{82} of the interaction scheme `leaveRegions` prevents another match for itself because it creates the situation forbidden by its application condition ac_{82}. It follows that the application of each of these four interaction schemes as long as possible terminates.

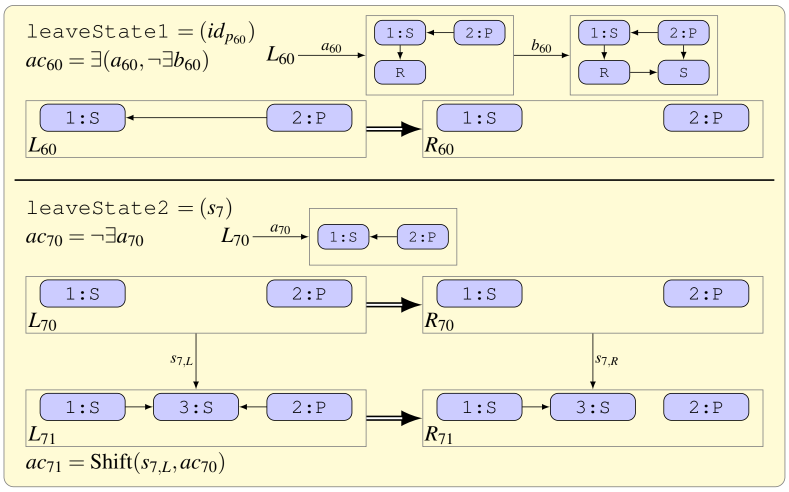
leaveState1 = $(id_{p_{60}})$
$ac_{60} = \exists(a_{60}, \neg\exists b_{60})$
L_{60}
a_{60}
b_{60}
1:S
2:P
R
S
R_{60}
leaveState2 = (s_7)
$ac_{70} = \neg\exists a_{70}$
L_{70}
a_{70}
R_{70}
$s_{7,L}$
$s_{7,R}$
3:S
L_{71}
R_{71}
$ac_{71} = \mathrm{Shift}(s_{7,L}, ac_{70})$

Figure 10: The interaction schemes `leaveState1` and `leaveState2`

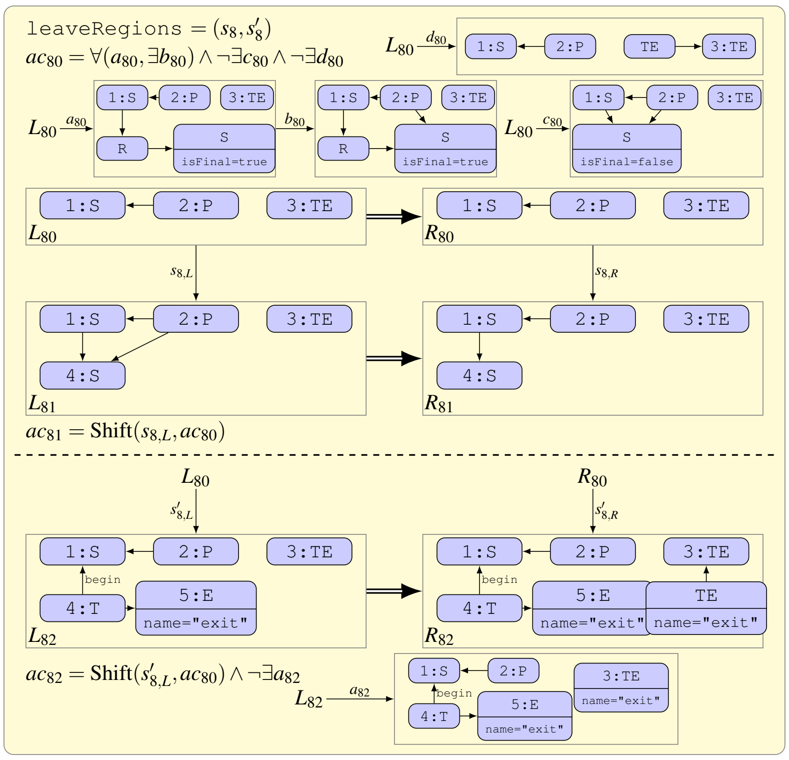
leaveRegions = (s_8, s'_8)
$ac_{80} = \forall(a_{80}, \exists b_{80}) \wedge \neg\exists c_{80} \wedge \neg\exists d_{80}$
L_{80}
d_{80}
a_{80}
b_{80}
c_{80}
1:S
2:P
3:TE
TE
R
S
isFinal=true
isFinal=false
R_{80}
$s_{8,L}$
$s_{8,R}$
4:S
L_{81}
R_{81}
$ac_{81} = \mathrm{Shift}(s_{8,L}, ac_{80})$
$s'_{8,L}$
$s'_{8,R}$
begin
4:T
5:E
name="exit"
L_{82}
R_{82}
$ac_{82} = \mathrm{Shift}(s'_{8,L}, ac_{80}) \wedge \neg\exists a_{82}$
a_{82}

Figure 11: The interaction scheme `leaveRegions`

Proposition 3 (Termination of semantical steps) *Given a well-behaved statechart, each semantical step terminates.*

Combining all the rules as explained above leads to the semantics of statecharts.

Definition 8 (Statechart semantics) The operational semantics of statecharts consists of one initialization step followed by as many as possible semantical steps defined as follows:

- *Initialization step.* For a statechart model $M \in VL_{SC}$ (see Definition 1) we obtain a model $M_{initial}$ by applying the sequence `setSub!`, `transSub!`, `init`, `enterRegions!` to M.
- *Semantical step.* Consider a model M_1 with M_1 obtained by a finite number of semantical steps from a model $M_{initial}$ for some $M \in VL_{SC}$, then a semantical step from M_1 to M_2 is computed by applying the sequence `transitionStep`, `enterRegions!`, `leaveState1!`, `leaveState2!`, `leaveRegions!` to M_1.

Moreover, combining our termination results we can conclude the termination of the statecharts semantics for well-behaved statecharts.

Theorem 2 (Termination of interpreter semantics) *For well-behaved statecharts with finite event queue, the interpreter semantics terminates.*

Proof. According to Proposition 2 and Proposition 3, each initialization step and each semantical step terminates. Moreover, each semantical step consumes an event from the event queue. If it triggers an action, the acyclic action-event graph ensures that there are only chains of events triggering actions, but no cycles, such that after the execution of this chain the number of elements in the event queue actually decreases. Thus, after finitely many semantical steps the event queue is empty and the operational semantics terminates. □

5 Application to the Running Example

We now consider an initialization and a semantical step in our statechart example from Figure 1. In the top of Figure 12, we show the incoming event queue as needed for our system run to be processed. Note that the actions that are triggered by state transitions do not occur here because they are started internally, while the other events have to be supplied from the outside. Thus, the internal events are supplied by the semantical rules themselves, while the external ones have to be given. For simplicity, we assume that the complete external event queue is given in advance, but the events could also appear one after the other using some additional rule that appends an event at the end of the queue. In the bottom of Figure 12, the current states and their corresponding state transitions are depicted. We want to simulate these semantical steps now using the rules for the semantics applied to the statechart in abstract syntax in Figure 4, extended by the event queue from Figure 12.

First, the initialization has to be done. We compute all `sub`-edges by applying the rules `setSub` and `transSub` in Figure 7 as long as possible. For the actual initialization, we apply

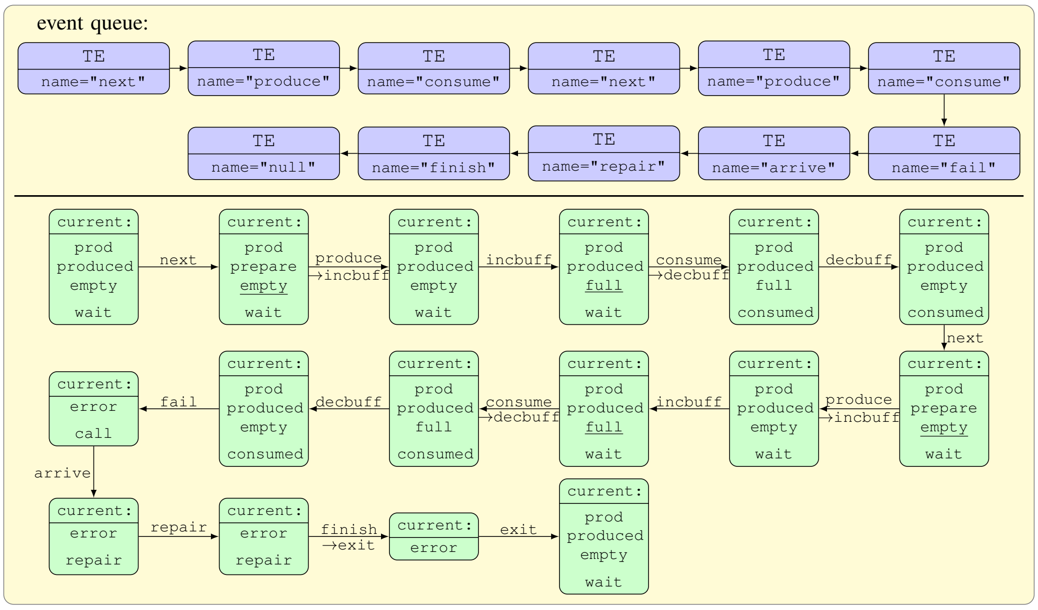

Figure 12: Event queue and state transitions

the interaction scheme `init` from Figure 8 followed by the application of `enterRegions` as long as possible. With `init`, we connect the state machine and the pointer node, and in addition set the pointer to the `prod`-state using a `new`-edge. Now the only available kernel match for `enterRegions` is the match mapping node `1` to the `prod`-state, and with maximal matchings we obtain the bundle of kernel morphisms $(id_{p_{40}}, s_4, s_4, s_4)$, where the node `4` in L_{41} is mapped to the states `produced`, `empty`, and `wait`, respectively. After the application of the corresponding amalgamated rule, the current pointer is now connected to the state machine and the state `prod`, and via `new`-edges to the states `produced`, `empty`, and `wait`. Further applications of `enterRegions` using these three states for the kernel matches, respectively, lead to the bundle $(id_{p_{40}})$ thus changing the `new`-edges to `current`-edges by its application. As a result, the states `prod`, `produced`, `empty`, and `wait` are current, which is the initial situation for the statemachine as shown in Figure 13, where the current states are marked by thicker lines. We do not find additional matches for `enterRegions`, as we only have one level of nesting in our diagram, which means that the initialization is completed.

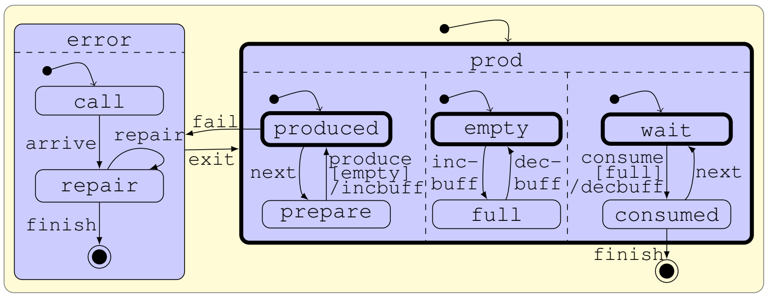

Figure 13: The statechart after the initialization step

For a state transition, the interaction scheme transitionStep in Figure 9 is applied, followed by the interaction schemes enterRegions!, leaveState1!, leaveState2!, and leaveRegions! given in Figure 8, 10, and 11.

For the initial situation, the kernel rule p_{50} in Figure 9 has to be matched such that the node 2 is mapped to the first trigger element next and the node 3 to produce, otherwise the application condition of the rule p_{50} would be violated. For the multi rules, there are two events with the name next, but since the state consumed is not current, only one match for L_{51} is found mapping the nodes 4 to the current state produced and 6 to the state prepare. All application conditions are fulfilled, since this transition does not have a guard or action, and the state produced does not have any substates. Thus, the application of the bundle (s_5) deletes the first trigger element next, which is done by the kernel rule, and redirects the current pointer from produced to prepare via a new-edge. An application of the interaction scheme enterRegions using the bundle $(id_{p_{40}})$ changes this new-edge to a current-edge. Since we do not find further matches for L_{40}, L_{60}, L_{71}, L_{81}, and L_{82}, the other interaction schemes cannot be applied. This means that the states prod, prepare, empty, and wait are now the current states, which is the situation after the state transition triggered by next as shown in Figure 12.

For the next match of the kernel rule p_{50}, the node 2 is mapped to the new next trigger element produce and 3 is mapped to consume. Since the transition produce has an action, we cannot apply the multi rule p_{51} but p_{52} has a valid match. In particular, the application condition is fulfilled because the guard condition state empty is current and the state prepare does not have any substates. Thus, the bundle (s'_5) leads to the deletion of the trigger element produce, the current pointer is redirected from prepare to produced, and a new trigger element incbuff is inserted with a next-edge to the trigger element consume. Again, enterRegions changes the new- to a current-edge and we do not find further matches for L_{40}, L_{60}, L_{71}, L_{81}, and L_{82}. This means that now the states prod, produced, empty, and wait are current.

We can process our trigger element queue step by step retracing the state transitions by the application of the rules. We do not explain all steps explicitly, but skip until after the last decbuff-trigger element, which leads to the current states prod, produced, empty, and consumed.

The next match of the kernel rule p_{50} maps the nodes 2 to the trigger element fail and 3 to arrive. The only match for the multi rules maps the nodes 4 and 6 in L_{51} to the states produced and error, respectively. Since the application condition is fulfilled, the application of the bundle (s_5) leads to the deletion of the trigger element fail, and the current pointer is redirected from produced to error. Now we find a match for the interaction scheme enterRegions mapping the node 1 to the state error and 4 to the state call. Thus the application of the bundle $(id_{p_{40}}, s_4)$ adds a new pointer to the state call, which is then changed from new to current. Afterwards, we find a match for leaveState1, where the kernel rule match maps the node 1 to the state prod. The application condition is fulfilled because there is a region - the one for the producer - where no state is current. Thus, the current-edge to prod is deleted. No more matches for L_{60} can be found, but there are two different matches for the multi rule p_{71} of leaveState2 matching the node 3 to the states empty and consumed, respectively. The application of the bundle (s_7, s_7) then leads to the deletion of the current pointer for the states empty and consumed. No more matches for L_{71}, L_{81}, and L_{82} can be found. Altogether, the states error and call are current now. This is exactly the situation as described in Figure 12 after the state transition triggered by the fail-event.

Now we skip again two more trigger elements leading to the remaining trigger element queue finish $\rightarrow$ null and the current states error and repair. The kernel rule p_{50} is now matched to these two trigger elements, and the application of the bundle (s_5) deletes the trigger element finish and redirects the current pointer from repair to final, the final state within the error-state. With enterRegions, the corresponding new-edge is set to current. No matches for L_{60} and L_{71} can be found, but we find a match for the interaction scheme leaveRegions, where the kernel rule is matched such that the node 1 is mapped to the state error and 3 is mapped to the null-trigger element. The application condition is fulfilled because all current substates of error are final states - actually, there is only the one - and null is the first trigger element in the queue. Now there is a match for L_{81} mapping the node 4 to the state final and a match for L_{82} mapping the nodes 4 and 5 to the transition and the event between the stated error and prod. After the application of the bundle (s_8, s'_8), the current pointer is deleted from the final-state, and a new exit-trigger element is inserted before the null-trigger element. No more matches for L_{81} and L_{82} can be found, thus only the state error is current. A last application of the interaction scheme transitionStep followed by enterRegions leads back to the initial situation and completes our example, since the event queue is empty except for the default element null.

According to Thm. 2, the simulation of our example terminates because our statechart is *well-behaved* and the event queue is finite.

6 Implementation

Recently, we have extended our tool Henshin[1] by visual editors for amalgamated rules and application conditions [BESW10]. Henshin is an Eclipse plug-in supporting visual modeling and execution of EMF model transformations, i.e. transformations of models conforming to a meta-model given in the EMF Ecore format. The transformation approach we use in our tool is based on graph transformation concepts which are lifted to EMF model transformation by also taking containment relations in meta-models into account [ABJ$^+$10].

The recent extensions of Henshin enable us to validate the model of the visual interpreter semantics presented in this paper. The startgraph of our statechart interpreter is modeled in Henshin as an EMF instance (see Figure 14).

For simulation, Henshin supports the definition of control structures (called transformation units) for rules, such as *"apply rule r_1 once, and then the rules from the set $\{r_1, r_2\}$ in arbitrary order and as long as possible"*. Transformation units may be nested, the atomic unit being a rule.

The main transformation unit for the statechart simulation is shown in Figure 15. Here, the initialization step is executed by applying the subunit *initStatechart* (see right part of Figure 15). This step is realized by a sequence of units inserting at first the auxiliary edges of the type *sub* in the statechart model by applying the CountedUnit *initSubEdges* (containing a rule) as

[1] http://www.eclipse.org/modeling/emft/henshin/

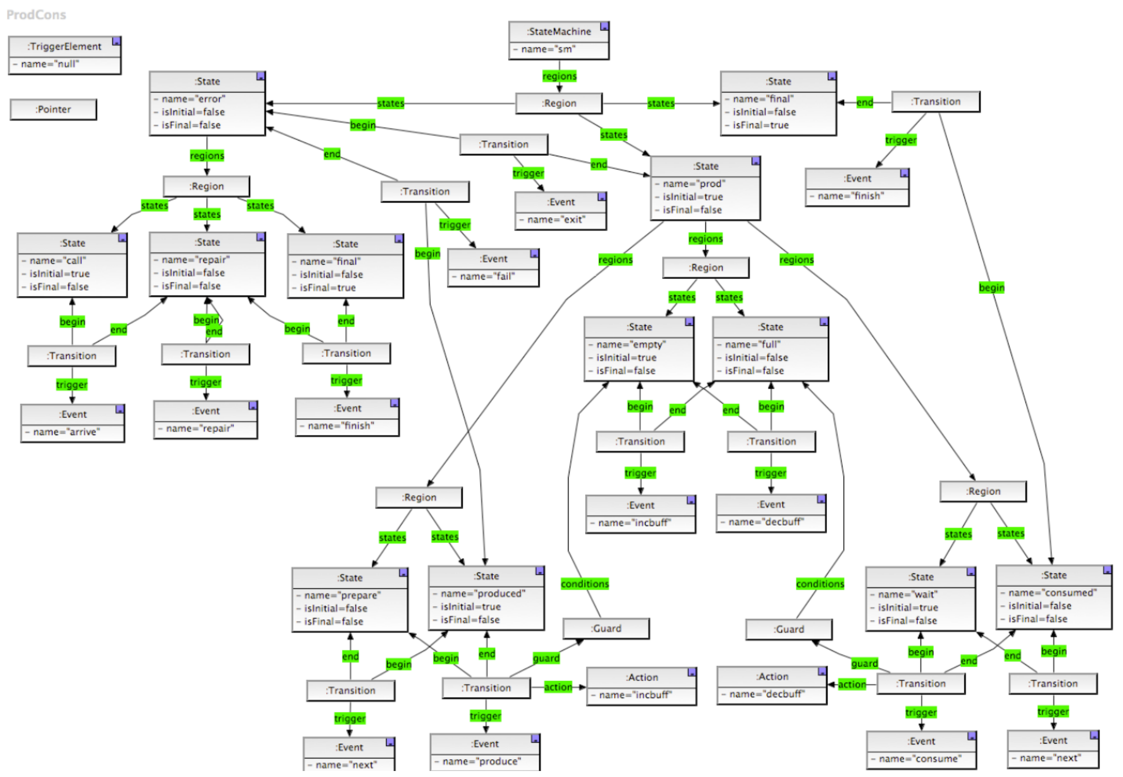

Figure 14: The initial statechart modeled in Henshin

long as possible (denoted by the count number "-1"), then applying the AmalgamationUnit *init* which corresponds to the interaction scheme *init* in Figure 8, followed by the interaction scheme *enterAllRegions* applied as long as possible.

Having performed the initialization step, the second step in the main unit *execute* consists of performing as many semantical steps as possible, triggered by the event queue. The unit *executeAllEvents* applies its subunit *executeEvent* (shown in the left part of Figure 16) as long as possible.

In this step, the interaction scheme *transitionStep* is applied, followed by as many applications as possible of the interaction schemes for entering regions and states, and leaving them afterwards. Interaction schemes like *transitionStep* are visualized in Henshin as a rule set containing one kernel rule and one or more multi-rules (see right part of Figure 16).

A multi-rule view in Henshin shows in an integrated way the corresponding kernel rule elements as simple rectangles and the additional multi-rule elements as rectangles with a shadow (see Figure 17). Application conditions for rules are visualized as logical connector blocks to the left of the rule's left-hand side (see left side of Figure 17), where the inherent morphisms can be expanded to a morphism view similar to the rule view, where mappings are indicated by colors and the numbers of the nodes.

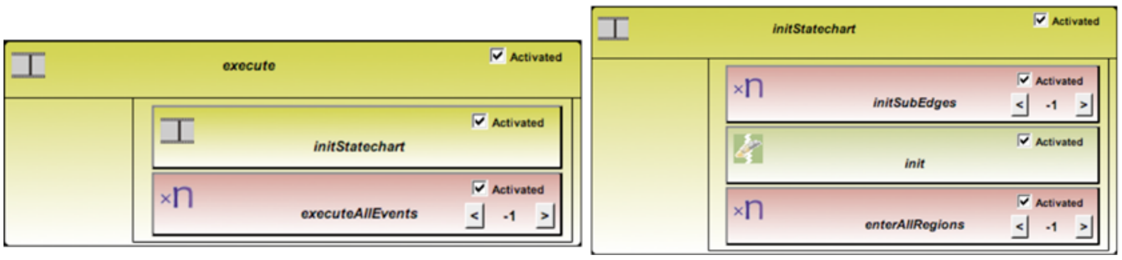

Figure 15: The main transformation unit *execute* (left) and its subunit *initStatechart* (right)

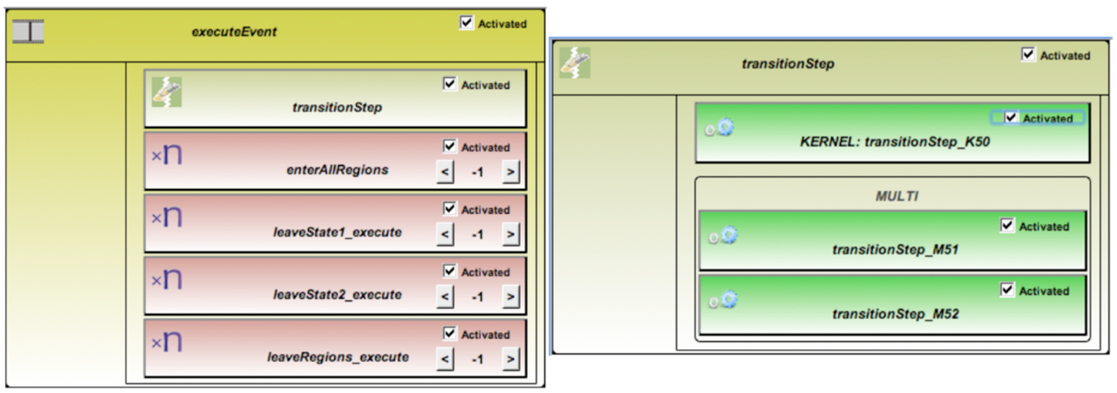

Figure 16: The transformation unit *executeEvent* (left) and its subunit *transitionStep* (right)

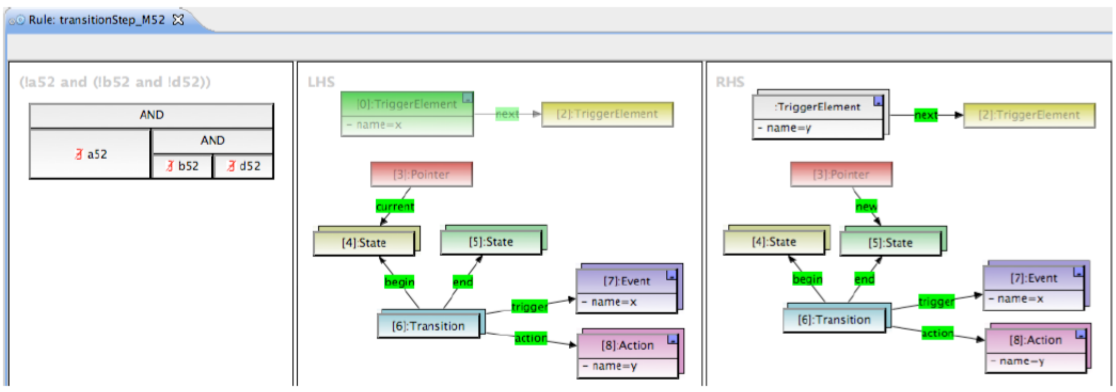

Figure 17: The multi-rule *transitionStep_M52*

7 Conclusion and Future Work

In this paper, we have defined a formal interpreter semantics for statecharts leading to a visual interpreter semantics. It is based on the theory of algebraic graph transformation and hence a solid basis for applying graph transformation-based analysis techniques. Unfortunately, the classical theory of graph transformations [Roz97] is not adequate to model the interpreter semantics of statecharts because we need rule schemes to handle an arbitrary number of transitions in orthogonal states in parallel. In this paper, we have solved this problem using amalgamated graph transformation [GEH10] in order to handle the interpreter semantics. As a first step towards the analysis of this semantics we have shown the termination of initialization and semantical steps and, more general, the termination of the interpreter semantics for *well-behaved* statecharts.

Our formal approach is also a promising basis to analyze other properties like confluence and functional behavior in the future. Since termination and local confluence implies confluence, it is sufficient to analyze local confluence. This has been done successfully for algebraic graph transformation based on standard rules and critical pairs [EEPT06]. It remains to extend this analysis from standard rules to amalgamated rules constructed by interaction schemes and to take into account maximal matchings as well as all essential amalgamated rules constructed from one interaction scheme.

The formal definition of syntax and operational semantics of statecharts in this paper provides the basis for a model transformation from statecharts to Petri nets [GEH11], which is shown to be semantics-preserving in [Gol11].

Another interesting research area to be considered in future is the nesting of kernel morphisms, which may lead to a hierarchical interaction scheme such that a semantical step of the statechart is actually a direct amalgamated transformation over one interaction scheme, and we no longer need rules for redirecting the `current` pointer afterwards.

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu et al. (eds.), *Proceedings of MOCELS 2010, Part I*. LNCS 6394, pp. 121–135. Springer, 2010.

[Bee02] M. Beeck. A Structured Operational Semantics for UML-statecharts. *Software and Systems Modeling* 1:130–141, 2002.

[BEE+10] E. Biermann, H. Ehrig, C. Ermel, U. Golas, G. Taentzer. Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In Engels et al. (eds.), *Graph Transformations and Model-Driven Engineering. Essays Dedicated to M. Nagl on the Occasion of his 65th Birthday*. LNCS 5765, pp. 121–140. Springer, 2010.

[BESW10] E. Biermann, C. Ermel, J. Schmidt, A. Warning. Visual Modeling of Controlled EMF Model Transformation using Henshin. *ECEASST* 32:1–13, 2010.

[BFH87] P. Böhm, H.-R. Fonio, A. Habel. Amalgamation of Graph Transformations: A Synchronization Mechanism. *JCSC* 34(2-3):377–408, 1987.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.

[EHL10] H. Ehrig, A. Habel, L. Lambers. Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. *ECEASST* 26:1–23, 2010.

[GEH10] U. Golas, H. Ehrig, A. Habel. Multi-Amalgamation in Adhesive Categories. In *Proceedings of ICGT 2010*. LNCS 6372, pp. 346–361. Springer, 2010.

[GEH11] U. Golas, H. Ehrig, F. Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. *ECEASST* 39:1–26, 2011. This volume.

[Gol11] U. Golas. *Analysis and Correctness of Algebraic Graph and Model Transformations*. PhD thesis, Technische Universität Berlin, 2011. Vieweg+Teubner.

[GP98] M. Gogolla, F. Parisi-Presicce. State Diagrams in UML: A Formal Semantics Using Graph Transformations. In *Proceedings of ICSE 1998*. Pp. 55–72. IEEE, 1998.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8:231–274, 1987.

[HP09] A. Habel, K.-H. Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *MSCS* 19(2):245–296, 2009.

[KGKK02] S. Kuske, M. Gogolla, R. Kollmann, H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler et al. (eds.), *Proceedings of IFM 2002*. LNCS 2335, pp. 11–28. Springer, 2002.

[Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Gogolla and Kobryn (eds.), *Proceedings of UML 2001*. LNCS 2185, pp. 241–256. Springer, 2001.

[MP96] A. Maggiolo-Schettini, A. Peron. A Graph Rewriting Framework for Statecharts Semantics. In Cuny et al. (eds.), *Graph Grammars and Their Application to Computer Science*. LNCS 1073, pp. 107–121. Springer, 1996.

[OMG09] OMG. Unified Modeling Language (OMG UML), Superstructure, Version 2.2. 2009.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, H. Hussmann. Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In Maibaum (ed.), *Fundamental Approaches to Software Engineering. Proceedings of FASE 2000*. LNCS 1783, pp. 127–146. Springer, 2000.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation - Formal Description and Application to Communication Based Systems*. PhD thesis, Technische Universität Berlin, 1996.

[TB94] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In Schneider and Ehrig (eds.), *Graph Transformations in Computer Science*. LNCS 776, pp. 380–394. Springer, 1994.

[Var02] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In Corradini et al. (eds.), *Proceedings of ICGT 2002*. LNCS 2505, pp. 378–392. Springer, 2002.