A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation

Ulrike Golas, Enrico Biermann, Hartmut Ehrig, and Claudia Ermel

Technische Universität Berlin, Germany
ugolas|enrico|ehrig|lieske@cs.tu-berlin.de

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 structures. Based on the existing visual semantics definitions, it is 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.

1 Introduction and Related Work

In [1], 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 [2], 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 [3], a structured operational semantics (SOS) for UML statecharts is given based on the preceding definition of a textual syntax for statecharts. The semantics uses Kripke structures and an auxiliary semantics using deduction, 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 [4], 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 [5], 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 [6], 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 [7], 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 [8], 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 again dependent on the concrete model, since the transformation rules have to be specified according to this model.

In [9], 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 structures encoding activation or conflicts of transitions and states.

The main advantage of our solution is that we do not need external control structures 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 [10] and give a short introduction to amalgamated transformation based on [11], 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. Finally, Section 6 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 Fig. 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.

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.

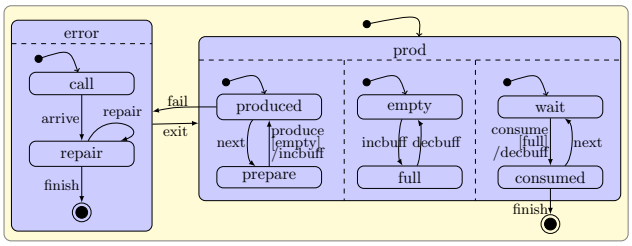


Fig. 1. Sample statechart `ProdCons`

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. Then, the machine has to be repaired before the `error`-state can be exited via the corresponding `exit`-transition and the standard behavior in the `prod`-state is executed again.

For our statechart language, we use typed attributed graphs, which are an extension of typed graphs by attributes [10]. We do not give details here, but use an intuitive approach to attribution, 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 Fig. 2. We use multiplicities to denote some constraints directly in the type graph. To obtain valid statechart models, some more constraints are needed which are described in the following.

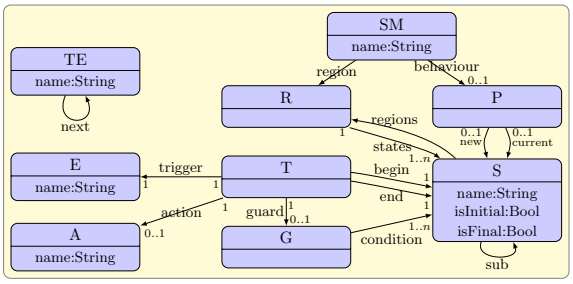


Fig. 2. Type graph TG_{SC} for statecharts

Each diagram consists of exactly one statemachine `SM` containing one or more regions `R`. A region contains states `S`, where state names are unique within each region. A state may again contain one or more regions. Each region is contained in either exactly one state or the statemachine. 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 final states cannot contain regions. 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 ore more

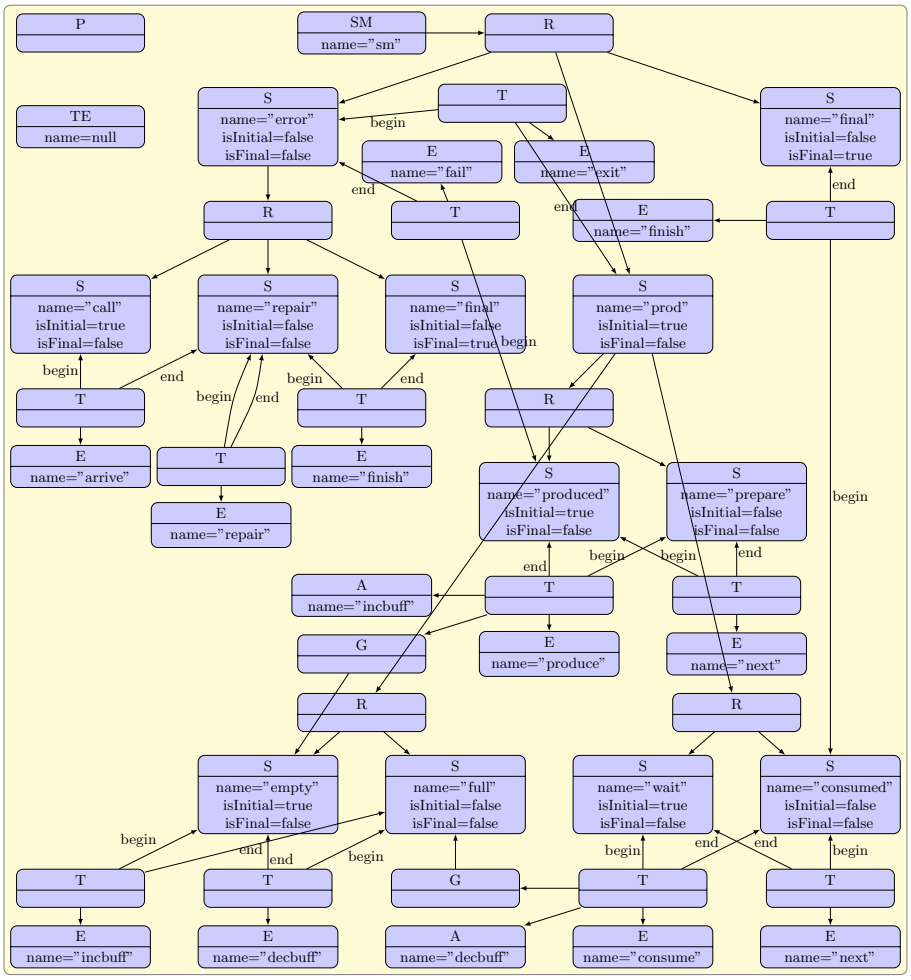


Fig. 3. Statechart `ProdCons` in abstract syntax

states as condition. There is a special event with attribute value `name="exit"` reserved for exiting a state after the completion of all its orthogonal regions, which cannot have a guard condition. Final states cannot be the beginning of a transition and their name has to be `"name=final"`. Transitions cannot link states in different orthogonal regions of the same superstate.

A pointer `P` describes the active states of the statemachine. Note that newly inserted current states are marked by the `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 is due to our semantics definition, 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

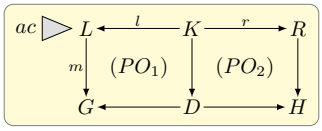
the events which have to be handled by the statemachine. Note that this is not necessarily a queue because of orthogonal states, but for simplicity we call it event queue. There are at least the empty trigger element with attribute value `name=null` and exactly one pointer in each diagram.

In Fig. 3, the sample statechart `ProdCons` from Fig. 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 Fig. 3) can be filled by events to be processed (see Fig. 9 in Section 5 for a possible event queue for our sample statechart).

3 Introduction to Amalgamated Graph Transformation

In this section, we review the basic ideas of algebraic graph transformation [10] and give a short introduction into amalgamated transformation based on [11], 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 $L \xrightarrow{m} G$ 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 below 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.



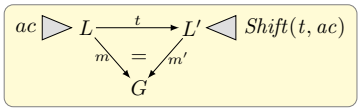
A graph transformation is a sequence of direct transformations, denoted by $G \overset{*}{\Longrightarrow} H$, and the graph language $L(GG)$ of graph grammar GG is the set $L(GG) = \{G \mid \exists\ SG \overset{*}{\Longrightarrow} G\}$ of all graphs G derivable from SG.

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 [12], 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 [13] 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, and amalgamation can be considered as a kind of "synchronized parallelism".

For the interpreter semantics of statecharts we need an extension of amalgamation in [13] 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 [10] instead of "plain graphs", and third, we need rules with application conditions.

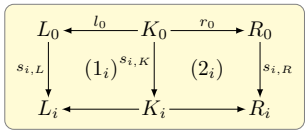
In the following, we formulate the extended amalgamation concept for a general notion of graphs and application conditions, where *general graphs* are objects in a weak adhesive HLR category [10] and *general application conditions* are nested application conditions [14], including positive and negative 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 [10] for more details). A match $L \xrightarrow{m} G$ satisfies a positive (negative) condition of the form $\exists a$ ($\neg\exists a$) for $L \xrightarrow{a} N$ if there is a (no) injective $q : N \to G$ with $q \circ a = n$. More general, $L \xrightarrow{m} G$ satisfies a nested condition of the form $\exists(a, ac_N)$ on L with condition ac_N on N if there is an injective $N \xrightarrow{q} G$ 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 Figs. 6 - 7).

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 [15].

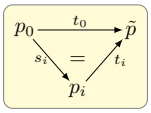


Based on [11], we are now able to introduce amalgamated rules and transformations with a common subrule p_0 of $p_1, \ldots, p_n$. A kernel morphism describes how the subrule is embedded into the larger rules.

Definition 1 (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.*



Definition 2 (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 $\tilde{p}$.*



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 Fig. 4 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 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 $i = 1, 2$.

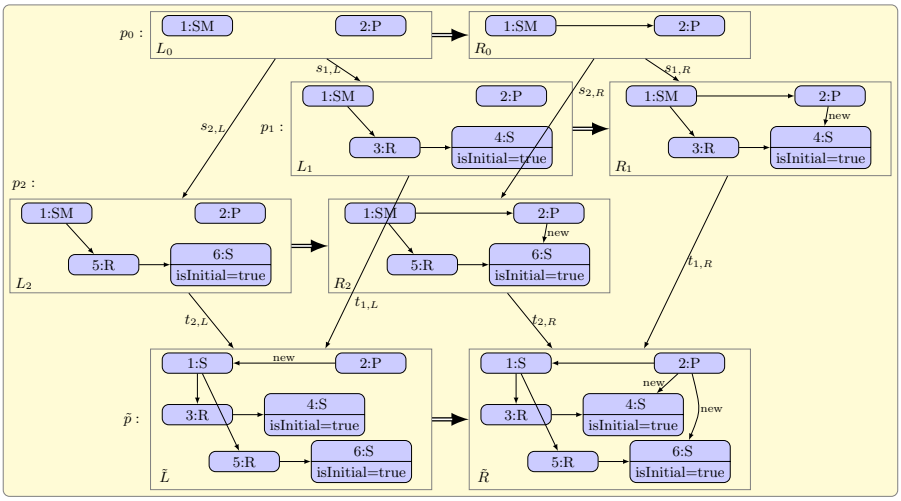


Fig. 4. Construction of amalgamated rule

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 [13] and leads to the following theorem [11].

Theorem 1 (Multi-amalgamation). *Given rules $p_0, \ldots, p_n$, where p_0 is a subrule of p_i, and multi-amalgamable direct transformations $G \overset{p_i,m_i}{\Longrightarrow} G_i$ $(i = 1, \ldots, n)$, 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 leading to

the amalgamated transformation because the componentwise gluing is a colimit construction. For the complete proof see [16].

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 $is = \{s_1, \ldots, s_k\}$ with kernel morphisms $s_i : p_0 \to p_j$ $(j = 1, \ldots, k)$ is given, which defines different bundles of kernel morphisms $s'_i : p_0 \to p'_i$ $(i = 1, \ldots, n)$ where each p'_i corresponds to some p_j for $j \leq k$.

Given an interaction scheme, we want to apply as many rules p_i 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. This leads to a bundle of kernel morphisms $s'_i : p_0 \to p'_i$ $(i = 1, \ldots, n)$ and a multi-amalgamable bundle of direct transformations $G \overset{p'_i, m'_i}{\Longrightarrow} G_i$. Since p_0 is a subrule of p'_i for $i = 1, \ldots, n$, because p'_i corresponds to some p_j in the interaction scheme, we can apply Thm. 1 leading to an amalgamated transformation $G \overset{\tilde{p}', \tilde{m}}{\Longrightarrow} H$, where $\tilde{p}'$ is the amalgamated rule of $p'_1, \ldots, p'_n$ 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 3 (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* $SG \overset{*}{\Longrightarrow} G$ *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. 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. 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 4 (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.*

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

For the *initialization step*, we provide a finite event queue and compute all substates of all states, which is not shown here. Then, the interaction scheme `init` is applied followed by the interaction scheme `enterRegions` applied as long as possible, which are depicted in Fig. 5. With `init`, the pointer is associated to the statemachine and all initial states of the statemachine's regions. The interaction

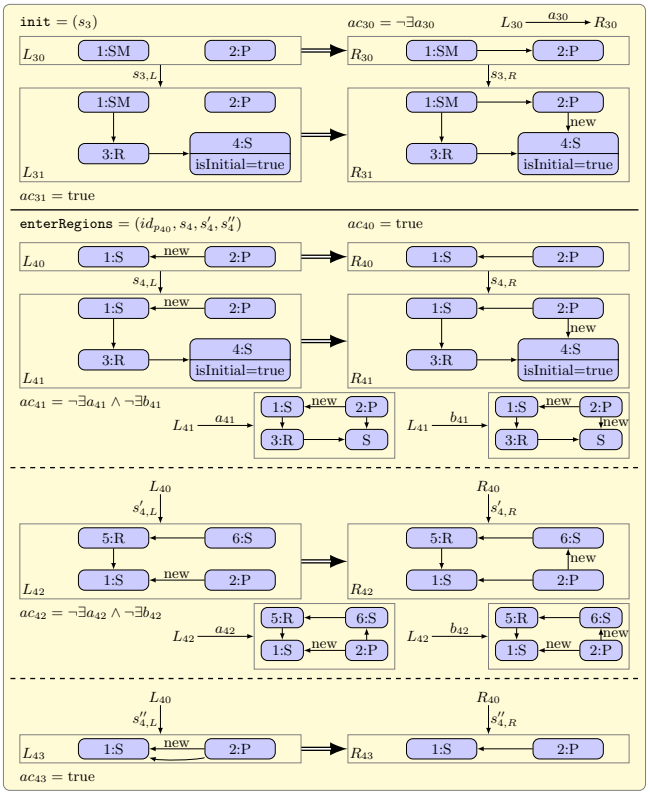


Fig. 5. The interaction schemes `init` and `enterRegions`

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, all substates are handled. Note that only those initial substates become active that are contained in a hierarchy of nested initial states. The interaction scheme `enterRegions` also contains the identical kernel morphism $id_{p_{40}} : p_{40} \to p_{40}$ to ensure that the kernel rule is also applied in the lowest hierarchy level. 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 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 in-

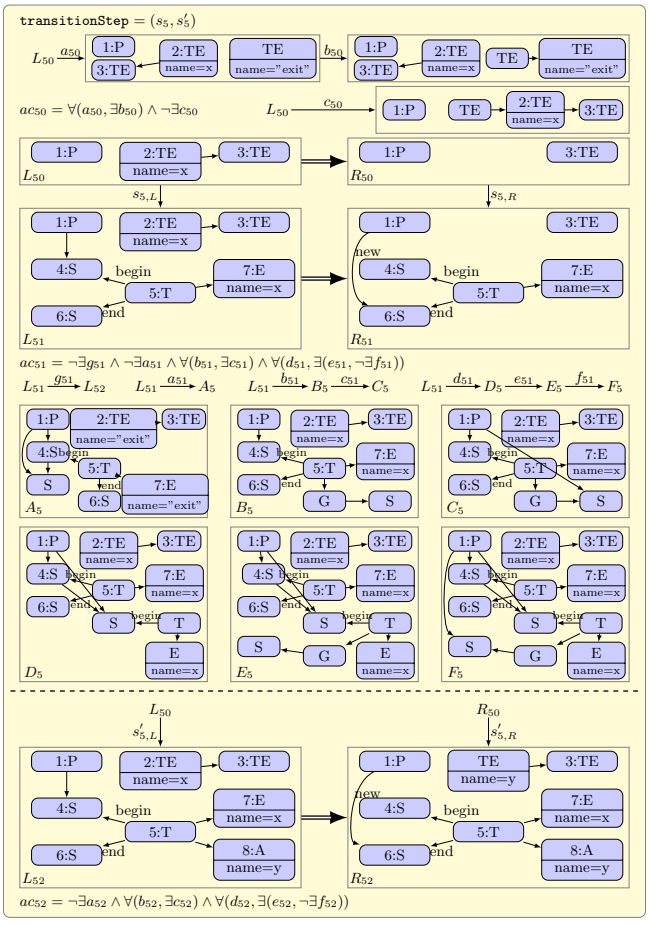


Fig. 6. The interaction scheme `transitionStep`

teraction 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.

Fact 1 (Termination of initialization step). *For well-behaved statecharts, the initialization step terminates.*

A *semantical step*, i.e. switching from one state to another, is done by applying the interaction scheme `transitionStep` shown in Fig. 6 followed by the interaction schemes `enterRegions!`, `leaveState1!`, `leaveState2!`, and `leaveRegions!` given in Fig. 5, Fig. 7, and Fig. 8 in this order, where `!` means that the corresponding interaction scheme is applied as long as possible.

For 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. `exit` trigger elements are handled with priority ensured by the application condition ac_{50}. 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 be the application conditions ac_{51} and ac_{52}. Moreover, if an action is provoked, it 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 $a_{51}, \ldots, f_{51}$ containing an additional 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`.

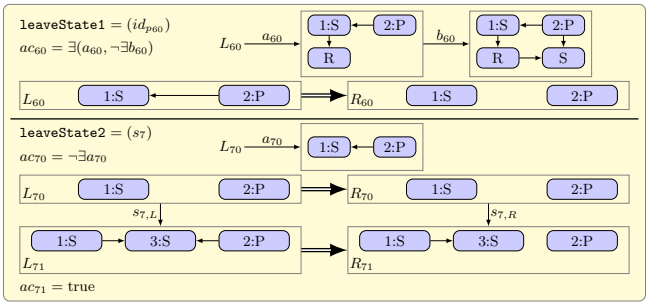


Fig. 7. The interaction schemes `leaveState1` and `leaveState2`

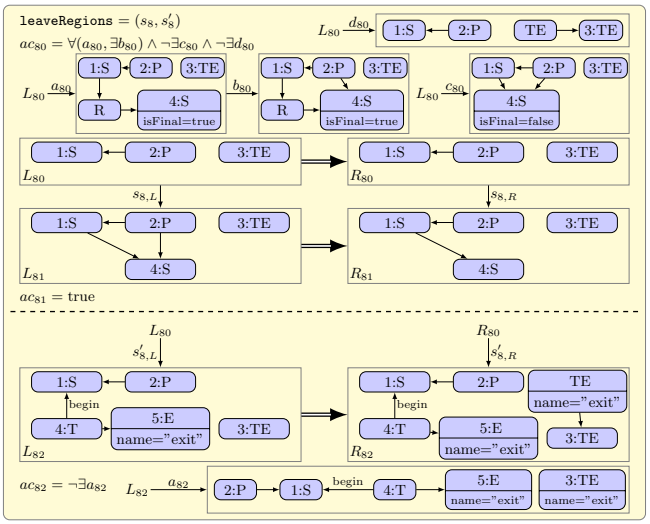


Fig. 8. The interaction scheme `leaveRegions`

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. The interaction scheme `enterRegions` terminates as shown in Fact 1. The interaction schemes `leaveState1`, `leaveState2` as well as the multi rule p_{81} of `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.

Fact 2 (Termination of semantical steps). *Given a well-behaved statechart, each semantical step terminates.*

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 Idea: Each initialization step and each semantical step terminates acc. to Facts 1 and 2. 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 interpreter semantics terminates.

5 Application to the Running Example

We now consider an initialization and a semantical step in our statechart example from Fig. 1. In the top of Fig. 9, we show an incoming event queue as needed for our system run to be processed. Note that the actions triggered by transitions do not occur here because they are started internally, while the other events have to be supplied from the environment. Below, the current states and their corresponding state transitions are depicted.

For simulation, we apply the rules for the semantics starting with the graph in abstract syntax in Fig. 3, extended by the event queue from Fig. 9 and all `sub`-edges marking that a state is a substate of its superstate.

For the initialization step, we apply the interaction scheme `init` from Fig. 5 followed by `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 node `4` in L_{41} is mapped to the states `produced`, `empty`, and `wait`, respectively. After applying the corresponding amalgamated rule, the current pointer is now connected to the state machine and state `prod`, and via `new` edges to the states `produced`, `empty`, and `wait`. Further applications of `enterRegions` using these three states for kernel matches, respectively, lead to the bundle $(id_{p_{40}})$, thus changing the `new` to `current` edges by its application. As result, the states `prod`, `produced`, `empty`, and `wait` are current, which is the initial situation for the statemachine as shown in Fig. 9. We do not find additional matches for `enterRegions` as we have only one level of nesting in our diagram, which means that the initialization is completed.

For a state transition, the interaction scheme `transitionStep` in Fig. 6 is applied, followed by the interaction schemes `enterRegions!`, `leaveState1!`, `leaveState2!`, and `leaveRegions!` given in Fig. 5, Fig. 7, and Fig. 8.

For the initial situation, the kernel rule p_{50} in Fig. 6 has to be matched such that node `2` is mapped to the first trigger element `next` and node `3` to `produce`,

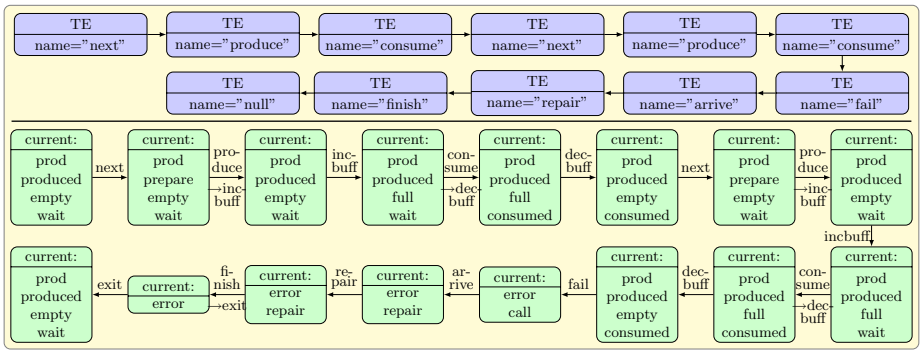


Fig. 9. Event queue and state transitions

otherwise the application condition of the rule would be violated. For the multi rules, there are two events of name `next`, but since the state `consumed` is not current, only one match for L_{51} is found mapping node `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` 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 Fig. 9. The procession of the remaining trigger elements works analogously.

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

6 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 [12] 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 [11] 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 [10]. 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.

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.

References

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

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

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

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

[5] Maggiolo-Schettini, A., Peron, A.: A Graph Rewriting Framework for Statecharts Semantics. In Cuny, J., Ehrig, H., Engels, G., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science. Volume 1073 of LNCS., Springer (1996) 107–121

[6] Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Gogolla, M., Kobryn, C., eds.: The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Proceedings of UML 2001. Volume 2185 of LNCS., Springer (2001) 241–256

[7] Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler, M., Petre, L., Sere, K., eds.: Integrated Formal Methods. Proceedings of IFM 2002. Volume 2335 of LNCS., Springer (2002) 11–28

[8] Gogolla, M., Parisi-Presicce, F.: State Diagrams in UML: A Formal Semantics Using Graph Transformations. In: Software Engineering. Proceedings of ICSE 1998, IEEE (1998) 55–72

[9] Varró, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G., eds.: Graph Transformation. Proceedings of ICGT 2002. Volume 2505 of LNCS., Springer (2002) 378–392

[10] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs. Springer (2006)

[11] Golas, U., Ehrig, H., Habel, A.: Multi-Amalgamation in Adhesive Categories. In: Proceedings of ICGT 2010. (2010) Accepted.

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

[13] Böhm, P., Fonio, H.R., Habel, A.: Amalgamation of Graph Transformations: A Synchronization Mechanism. Journal of Computer and System Sciences **34**(2-3) (1987) 377–408

[14] Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Mathematical Structures in Computer Science **19**(2) (2009) 245–296

[15] Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. Electronic Communications of the EASST (2010) To appear.

[16] Golas, U.: Multi-Amalgamation in M-Adhesive Categories: Long Version. Technical Report 2010/05, Technische Universität Berlin (2010)