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)

Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions

Ulrike Golas, Hartmut Ehrig and Frank Hermann

26 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

Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions

Ulrike Golas[1], Hartmut Ehrig[2] and Frank Hermann[23]

[1] golas@zib.de, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany
[2] ehrig|frank@cs.tu-berlin.de, Technische Universität Berlin, Germany
[3] frank.hermann@uni.lu, SnT, Université du Luxembourg, Luxembourg

Abstract: Triple graph grammars are a successful approach to describe exogenous model transformations, i.e. transformations between models conforming to different meta-models. Source and target models are related by some connection part, triple rules describe the simultaneous construction of these parts, and forward and backward rules can be derived modeling the forward and backward model transformations. As shown already for the specification of visual models by typed attributed graph transformation, the expressiveness of the approach can be enhanced significantly by using application conditions, which are known to be equivalent to first order logic on graphs.

In this paper, we extend triple rules with a specific form of application conditions, which enhance the expressiveness of formal specifications for model transformations. We show how to extend results concerning information preservation, termination, correctness, and completeness of model transformations to the case with application conditions. We illustrate our approach and results with a model transformation from statecharts to Petri nets.

Keywords: model transformation, triple graph grammar, application condition

1 Introduction

Specification of models and model transformations play a central role in model-driven software development. For the specification of visual models and languages, it is common practice to use UML modeling techniques for the concrete syntax with underlying typed attributed graph transformation for the abstract syntax. The visual language can be defined in a declarative way by a meta-model with OCL-constraints or – on the abstract level – by a type graph and suitable graph constraints. Alternatively, the visual language can be generated on the abstract level by typed attributed graph grammars [EEPT06]. It is well-known that the expressiveness of such generative approaches can be enhanced by using graph grammar rules with negative application conditions (NACs), or even more by using nested application conditions in the sense of [HP09], which are known to be equivalent to first order logic on graphs and more expressive than NACs.

Graph transformation is a suitable approach to define model transformations [TEG+05]. Especially for exogenous model transformations, triple graph grammars (TGGs) [Sch94] are a well-suited formalism [SK08] and they were successfully applied in several domains [KS06, GL06a, GL06b]. Formal properties concerning information preservation, termination, correctness, and

completeness of model transformations have been studied already in [EEH08, EEHP09] based on triple rules without NACs, where the decomposition and composition theorem for triple graph transformation sequences in [EEE+07] plays a fundamental role. In [EHS09], this theorem has been extended to triple rules with NACs, but not yet to nested application conditions [HP09].

It is the main aim of this paper to extend the theory of model transformations based on TGGs to rules with nested application conditions, short application conditions, in order to enhance the expressiveness of model transformations including the generation of the source and target languages by corresponding source and target rules. We show that the decomposition and composition theorem can be extended to rules with application conditions. This allows to enhance the expressiveness of model transformations and to extend termination, correctness, completeness, and backward information preservation to this more general framework.

As a case study, we consider a model transformation from statecharts to Petri nets, where we use a combination of positive and negative application conditions and boolean operators as available in the framework of general application conditions, but not in the more restrictive framework of NACs. In our example, only one level of nesting is sufficient to model the necessary conditions, but the theory is developed for more complex situations as introduced in [Gol11].

This paper is organized as follows. In Section 2, we review triple rules and application conditions. We illustrate our approach with a model transformation from statecharts to Petri nets in Section 3. This case study is used as illustrating example in Section 4, where we define model transformations based on TGGs with application conditions leading to termination, correctness, completeness, and backward information preservation. A conclusion including related and future work is presented in Section 5.

We assume the reader to be familiar with the basics of UML statecharts [OMG09], Petri nets [Pet80], and graph transformation in the double pushout approach [EEPT06].

2 Review of Triple Graph Grammars and Application Conditions

Triple graph grammars [Sch94] are a well known approach for bidirectional model transformations. In [KS06], the basic concepts of triple graph grammars are formalized in a set-theoretical way, which is generalized and extended in [EEE+07] to typed, attributed graphs.

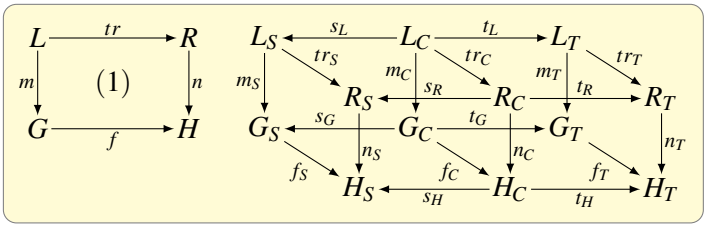
A *triple graph* $G = (G_S \xleftarrow{s_G} G_C \xrightarrow{t_G} G_T)$ consists of graphs G_S, G_C, and G_T, called source, connection, and target component, and two graph morphisms s_G and t_G mapping the connection to the source and target components. A triple graph morphism $f : G_1 \to G_2$ matches the single components and preserves the connection part.

The typing of a triple graph is done in the same way as for standard graphs via a type graph TG - in this case a triple type graph - and a typing morphism $type_G$ from the graph G into this type graph leading to the *typed triple graph* $(G, type_G)$. A typed triple graph morphism $f : (G_1, type_{G_1}) \to (G_2, type_{G_2})$ is a triple graph morphism f such that $type_{G_2} \circ f = type_{G_1}$.

Triple graphs and typed triple graphs, together with the component-wise compositions and identities, form the categories **TripleGraphs** and **Triple-Graphs$_{\mathbf{TG}}$**. When speaking of triple graphs, we consider both triple graphs and typed triple graphs, but do not explicitly mention the typing. Moreover, we define the morphism class $\mathscr{M}$ of injective triple graph morphisms which is used throughout the paper. Using this class $\mathscr{M}$, both categories can be extended to weak

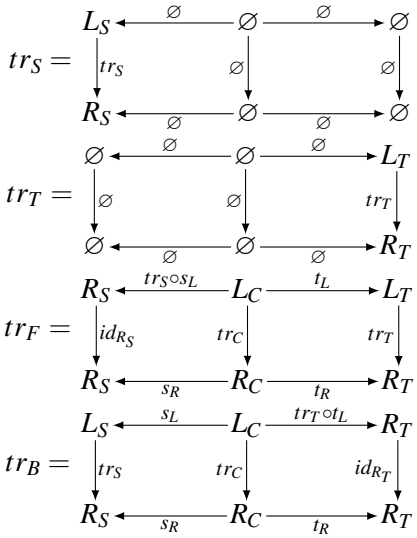
adhesive HLR categories [EEPT06] which allows us to instantiate the theory to transformations of triple graphs. The categorical foundations of weak adhesive HLR categories are not essential to understand this paper.

A *triple rule* $tr = (L \overset{tr}{\to} R)$ consists of triple graphs L and R, and an $\mathscr{M}$-morphism $tr : L \to R$. Since triple rules are non-deleting, we do not need a span of morphisms for a rule. A *direct triple transformation* $G \overset{tr,m}{\Longrightarrow} H$ of a triple graph G via a triple rule tr and a match $m : L \to G$ is given by the pushout (1), which is constructed as the component-wise pushouts in the S-, C-, and T-components, where the morphisms s_H and t_H are induced by the pushout of the connection component. Note, that due to the structure of the triple rules, double and single pushout approach are equivalent in this case.

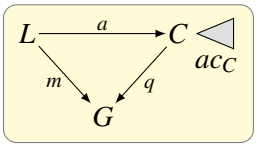


A *triple graph transformation system* $TGS = (TR)$ is based on triple graphs with a set TR of rules over them. A *triple graph grammar* $TGG = (TR, S)$ contains in addition a triple start graph S. For triple graph grammars, the generated language is defined by $VL = \{G \mid \exists \text{ triple transformation } S \overset{*}{\Rightarrow} G \text{ via rules in } TR\}$. Moreover, the source language $VL_S = \{G_S \mid (G_S \overset{s_G}{\leftarrow} G_C \overset{t_G}{\to} G_T) \in VL\}$ contains all standard graphs that are the source component of a derived triple graph. Similarly, the target language $VL_T = \{G_T \mid (G_S \overset{s_G}{\leftarrow} G_C \overset{t_G}{\to} G_T) \in VL\}$ contains all derivable target components.

From a triple rule, we can derive a source rule tr_S and a target rule tr_T, which specify the changes done by this rule in the source and target components, respectively. Moreover, the forward rule tr_F and the backward rule tr_B describe the changes done by the rule to the connection and target resp. source parts, assuming that the source resp. target rules have been applied already. Intuitively, the source rule creates a source model, which can then be transformed by the forward rules into the corresponding target model. This means that the forward rules define the actual model transformation from source to target models. Vice versa, the target rules create the target model, which can then be transformed into a source model applying the backward rules. Thus, the backward rules define the backward model transformation from target to source models.



An important extension is the use of rules with suitable *application conditions* as done in the next sections. Simple variants are positive application conditions of the form $\exists a$ for a morphism $a : L \to C$, demanding a certain structure in addition to L, and negative application conditions $\neg\exists a$, forbidding such a structure. A match $m : L \to G$ satisfies $\exists a$ ($\neg\exists a$) if there is a (no) $\mathscr{M}$-morphism $q : C \to G$ satisfying $q \circ a = m$. In more detail, we use nested application conditions [HP09], short application conditions, which are defined recursively. The application condition

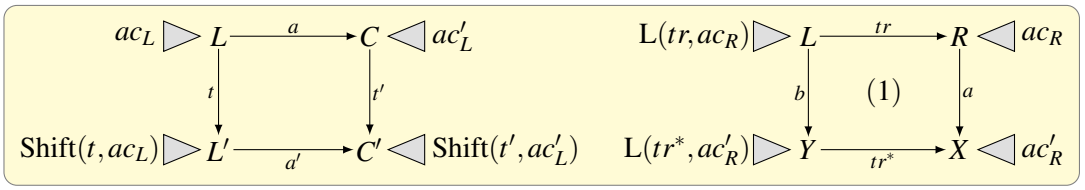


true is always satisfied. A more complex application condition $\exists(a, ac_C)$ on L consists of a morphism $a : L \to C$ and an application condition ac_C on C. For satisfaction, in addition to the existence of q it is required that q satisfies ac_C. Moreover, application conditions are closed under boolean operations. We use $\exists a$ as a short notion for $\exists(a, \text{true})$ and false for $\neg\text{true}$. In general, we write $m \models \exists(a, ac_C)$ if m satisfies $\exists(a, ac_C)$, and $ac_C \cong ac'_C$ denotes the semantical equivalence of ac_C and ac'_C on C. In the diagrams, an application condition ac_C on C is depicted by a triangle pointing towards C.

In order to handle rules with application conditions there are two important concepts, called the shift of application conditions over morphisms and morphism spans ([HP09, EHL10]):

1. Given an application condition ac_L on L and a morphism $t : L \to L'$ then there is an application condition $\text{Shift}(t, ac_L)$ on L' such that for all $m' : L' \to G$ holds: $m' \models \text{Shift}(t, ac_L) \iff m = m' \circ t \models ac_L$.

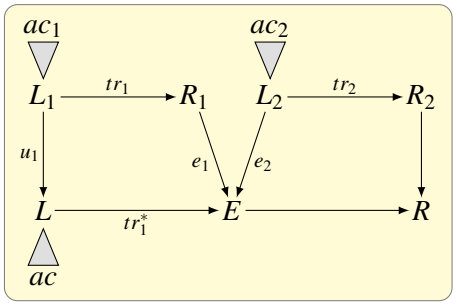
   For $ac_L = \exists(a, ac'_L)$ we define $\text{Shift}(t, ac_L) = \vee_{(a',t') \in \mathscr{F}} \exists(a', \text{Shift}(t', ac'_L))$ with $\mathscr{F} = \{(a', t') \mid (a', t') \text{ jointly epimorphic}, t' \in \mathscr{M}, t' \circ a = a' \circ t\}$,



2. Given a triple rule $tr = (L \xrightarrow{tr} R)$ and an application condition ac_R on R then there is an application condition $\text{L}(tr, ac_R)$ on L such that for all transformations $G \overset{tr,m}{\Longrightarrow} H$ with comatch n holds: $m \models \text{L}(tr, ac_R) \iff n \models ac_R$.

   For $ac_R = \exists(a, ac'_R)$ we define $\text{L}(tr, ac_R) = \exists(b, \text{L}(tr^*, ac'_R))$ if $a \circ tr$ has a pushout complement (1) leading to tr^*, and $\text{L}(tr, \exists(a, ac'_R)) = \text{false}$ otherwise.

One of the main results for graph transformation needed in this paper is the Concurrency Theorem, which is concerned with the execution of transformations which may be sequentially dependent. Given an arbitrary sequence $G \overset{tr_1,m_1}{\Longrightarrow} H \overset{tr_2,m_2}{\Longrightarrow} G'$ of direct transformations it is possible to construct an E-concurrent rule $tr_1 *_E tr_2 = (L \to R, ac)$. The object E is a jointly surjective overlap of R_1 and L_2. The construction of the concurrent application condition $ac = \text{Shift}(u_1, ac_1) \wedge \text{L}(p^*, \text{Shift}(e_2, ac_2))$ and $p^* = (L \xleftarrow{s_1} C_1 \xrightarrow{t_1} E)$ is again based on the two shift constructions. The Concurrency Theorem states that for the transformation $G \overset{tr_1,m_1}{\Longrightarrow} H \overset{tr_2,m_2}{\Longrightarrow} G'$ the E-concurrent rule $tr_1 *_E tr_2$ allows us to construct a direct transformation $G \overset{tr_1 *_E tr_2}{\Longrightarrow} G'$ via $tr_1 *_E tr_2$, and vice versa, each direct transformation $tr_1 *_E tr_2$ can be sequentialized.



3 Model Transformation from Statecharts to Petri Nets

In this section, we define a model transformation from a variant of UML statecharts [OMG09] to Petri nets [Pet80] using triple rules and application conditions. Statecharts may have orthogonal regions as well as state nesting. As a small restriction, we do not handle entry and exit actions, do not allow extended state variables, allow guards only to be conditions over active states, and allow only a depth of two for hierarchies of states. For the target language of Petri nets, we use nets with inhibitor arcs, contextual arcs, and open places. A transition with an inhibitor arc from a place (denoted by a filled dot instead of an arrow head) is only enabled if there is no token on this place. A contextual arc between a place and a transition (denoted by an edge without arrow heads), also known as read arc in the literature, means that this token is required for firing, but remains on the place. Open places allow the interaction with the environment, i.e. token may appear or disappear without firing a transition within the net. We assume all places to be open. With these restrictions for statecharts and extensions for Petri nets we are able to define a model transformation from statecharts to Petri nets which preserves the semantical behavior, at least on an informal level.

In Figure 1, the statechart `ProdCons` is depicted modeling a producer-consumer system. The whole state machine contains one region with the states `prod`, `error`, and a final state. 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 actually 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.

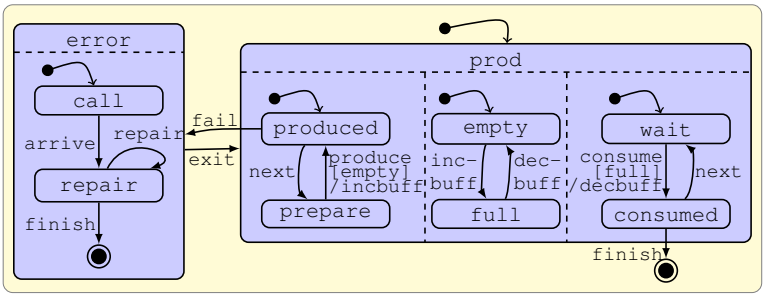


Figure 1: The example statechart in concrete syntax

Two possible events may happen causing a state transition leaving 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 the modeling, 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 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. Note, that for the typing of the edges we omit the edge types if they are clear from the node types they are connecting.

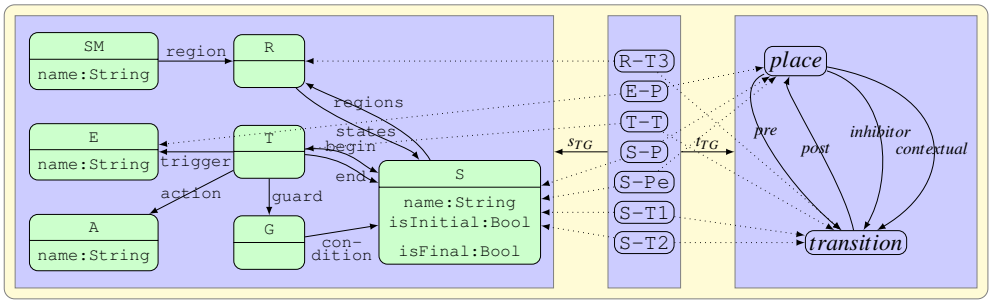


Figure 2: The triple type graph TG

In Figure 2, the triple type graph TG is depicted, containing in the left the source component of statecharts in abstract syntax, in the right the target component of Petri nets, and the connection component inbetween. To obtain valid statechart models, some constraints are needed which are described in the following but are not shown explicitly.

Each diagram consists of exactly one state machine `SM` containing one or more regions `R`. A region contains states `S`, where state names are unique within one region. A state may again contain one or more regions. Each region is contained in either exactly one state or the state machine. 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 final states cannot contain regions. 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. Moreover, final states cannot be the beginning of a transition and their name attribute has to be set to `name="final"`. In addition, transitions cannot link states in different orthogonal regions, which means that both regions are directly contained in the same state.

The language VL_{SC} consists of all typed attributed graphs respecting the source component TG_S of the type graph TG and the constraints as described above.

In the following, we present the triple rules that create simultaneously the statechart model, the connection part, and the corresponding Petri net. For simplicity, we depict the Petri nets in the target component in concrete syntax, while only writing node names in the connection component.

In general, each state of the statechart model is connected to a place in the Petri net, where a token on it represents that this state is current. Transitions between states are mapped to Petri net transitions and fire when the corresponding state transition occurs. Events are also connected to places, where all events with the same name share the same Petri net place. They are connected via a contextual arc to their corresponding transition

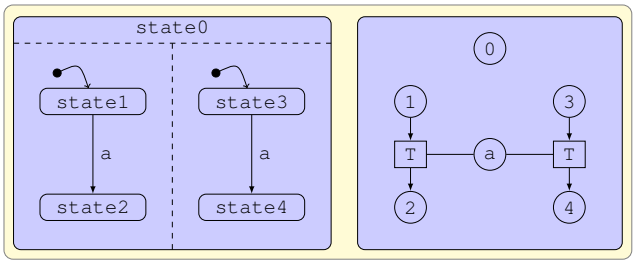


Figure 3: The basic correspondences

thus enabling the simultaneous firing of all enabled Petri net transitions when a token is placed there. By using contextual arcs it is possible that all transitions connected to an event with this name are enabled simultaneously if also their other pre-places are marked. Otherwise, we would not be able to fire all these transitions concurrently. They would not be independent but compete for the token. For independence, we had to know in advance how many of these transitions will fire to allocate that number of tokens on the event's place. For a guard, the Petri net transition of its transition in the statechart diagram is the target of a contextual arc from the place connected to the condition. Thus, we check also in the Petri net that this guard condition is fulfilled, i.e. the corresponding place holds a token, before firing the transition. Such a basic situation is depicted in Figure 3, where altogether five states and their corresponding places, two transitions – both in the statechart and the Petri net – and an event `a` with its corresponding place are shown. The hierarchy of the statechart is flattened, since Petri nets do not have such a concept. Note that all place are open places.

Additional places and transitions make sure that the effects of a state transition concerning involved sub- or superstates can be simulated also in the Petri net part. Each substate is connected via `S-T2` to a `T2`-transition which is the target of a pre-arc from its superstate. This makes sure that, when a state transition leaves this superstate, also all substates are no longer current. Each region within a state is connected via `R-T3` to a `T3`-transition which makes sure that, when no state inside this region is current, also the superstate is deactivated. These two situations are depicted in the example models in the top of Figure 4. Each state that may contain regions is connected via `S-T1` to a `T1`-transition that is the target of pre-arcs from all places of final states and inhibitor arcs from all other places in its regions, while the superstate's place is a contextual place as shown in the bottom of Figure 4. This makes sure that, when all substates are final, these substates are no longer current and, if it exists, the `exit`-action of the superstate can be initiated. For the handling of the special `"exit"`-events, each state which may be a superstate is connected via `S-Pe` to an `e`-place which handles the proper execution of this event regarding `T1`-

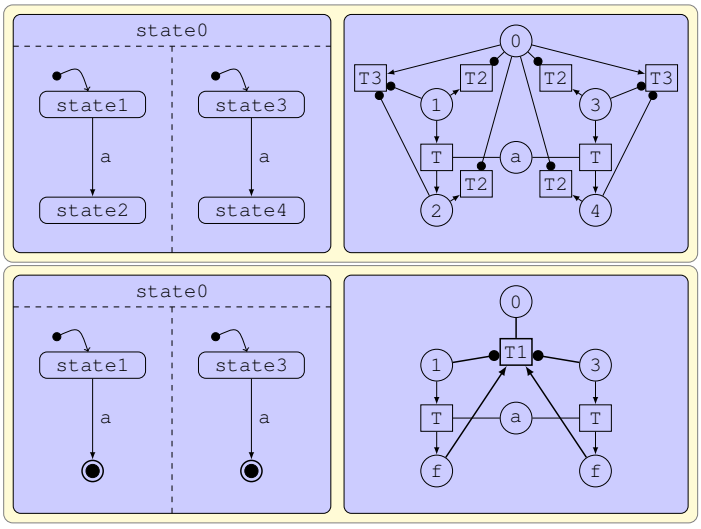


Figure 4: The additional correspondences

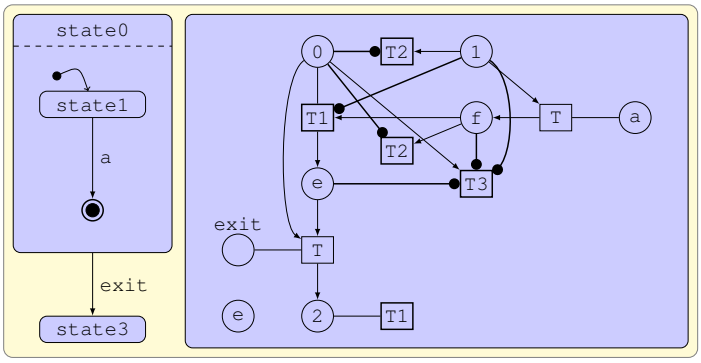


Figure 5: The handling of `exit`-events

and T3-transitions. The idea behind this place is, that when all final states are reached and an exit transition has to be invoked, the T1-transition delivers a token to the e-place which than triggers the execution of the transition in the Petri net as shown in Figure 5.

For the operational semantics, all places in the Petri net corresponding to currently active states will be marked. Depending on the semantical steps in the statechart, the open places in the Petri net produce and delete tokens. For example, triggering an external event in the statechart leads to a token on the events's place in the Petri net. Also for the handling of the hierarchical (de)activation the proper open places may fire triggered by the corresponding semantical rules for the statecharts. For example, when entering a state it's initial substates become active. This has to be handled in the Petri net by firing the corresponding open places. Thus, the Petri net for itself shows different semantical behavior than the statechart, since arbitrary firing of open places leads to strange behavior, but every semantical step in the statechart can be simulated by the Petri net. The rules for the operational semantics of statecharts are given in [GBEE11].

The start graph is the empty graph, and the first rule to be applied is the triple rule start shown in Figure 6 which creates the start graph of statecharts in the source component, and empty connection and target components. The application condition ac_0 is a so-called negative application condition (NAC) and forbids that the right hand side of the rule already exists before the rule is applied. Since a statemachine has exactly one node SM, the NAC ensures that the rule cannot be applied twice.

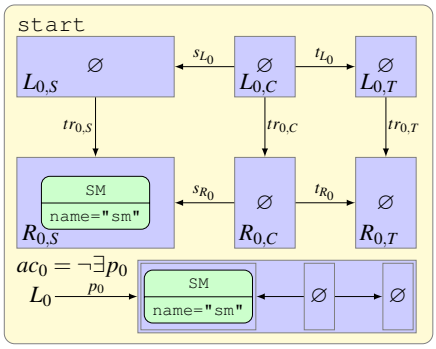


Figure 6: The rule start

In Figure 7, the triple rules newRegionSM and newRegionS are depicted which allow to create a new region of a state machine or of a state, respectively. Since each region has to have an initial state, this initial state is also created and connected to its corresponding place via S-P. With newRegionSM, the initial state is also connected to a T1-transition in the target component and another place via S-Pe. Moreover, if the new region is created inside a state by newRegionS the substate is the inhibitor of the superstate's T1-transition, the superstate inhibits a new T2-transition and the region and the substate inhibit a new T3-transition. For the triple rule newRegionS, the application condition forbids that the superstate is final or already a substate of another state. newRegionSM has the application condition true which is not depicted. Note that we allow parameters for the rules to define the attributes. Thus, the user has to declare the name of the newly created state when applying these triple rules.

In Figs. 8 and 9, the triple rules for creating new states are shown. With newStateSM and newStateS, new states inside a region of the state machine or of a state are created, which are not final states. Similarly, final states are created by the triple rules newFinalStateSM and newFinalStateS. In all cases, a corresponding place is created in the target component. As in the case of a new region, if creating a state as a substate of another state, there is a new T2-transition with this superstate as inhibitor and the new place inhibits the region's T3-transition. In case of a non-final substate, this substate inhibits the T1-transition of the superstate, whereas a final state within a state has to be connected to this superstate's T1-transition as a pre place. The application conditions of these rules make sure that the new state name is unique within its region and that, for final states, only one final state per region is allowed.

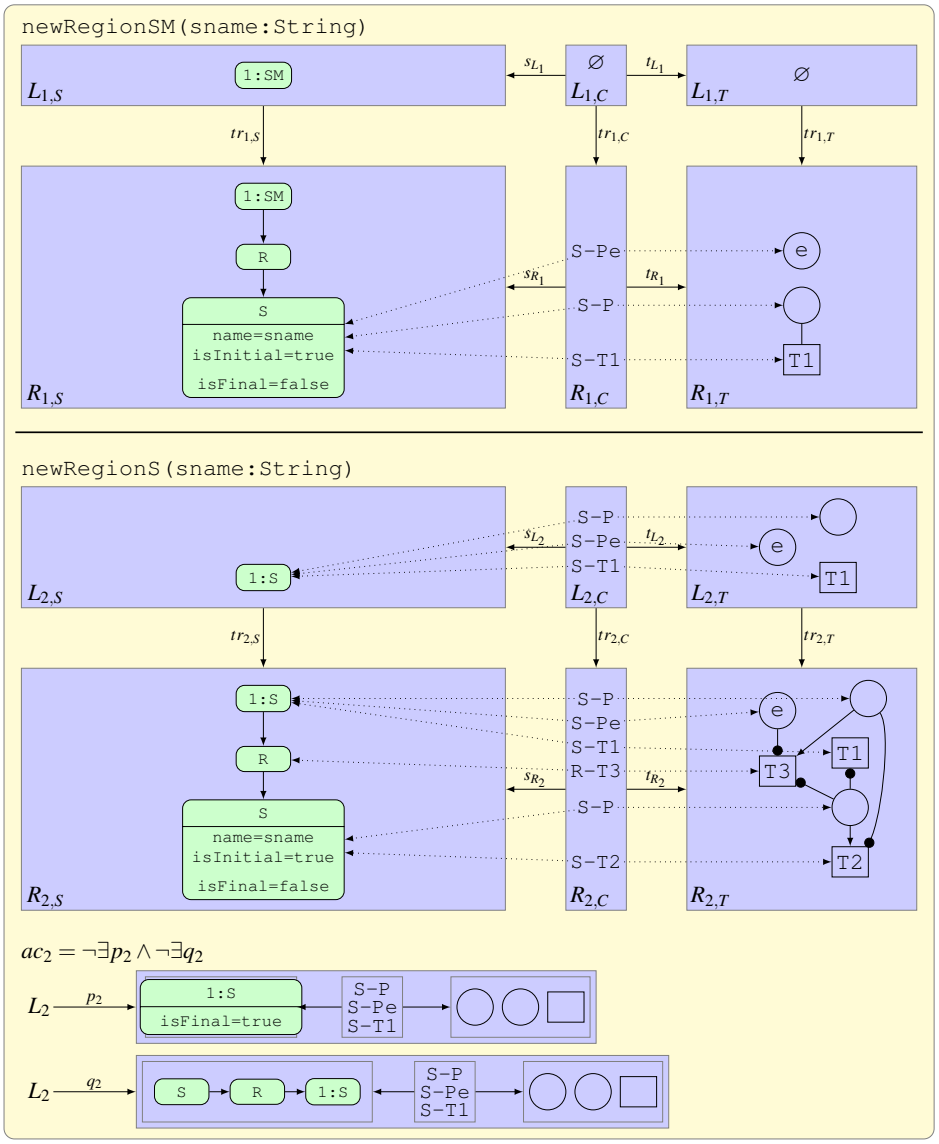


Figure 7: The triple rules `newRegionSM` and `newRegionS`

For the creation of a new transition, the triple rules `newTransitionNewEvent`, `newTransitionNewExit`, `newTransitionOldEvent`, and `newTransitionOldExit` in Figure 10 and Figure 11 are used. A new transition in the source part connected with a new Petri net transition in the target part is created, and in case of a new event, this event is connected with a new place which is a contextual place for the transition. Otherwise, the transition is connected with the place of the already existing event. In case of an `exit`-event, the place connected via `S-Pe` to the `begin`-state has to be connected to the new transition and the `begin`-state's `T1`-transition. The application conditions forbid that the `begin`-state is a final state and that states over different regions are connected by a transition (r_7, s_7, t_7, u_7), and ensure

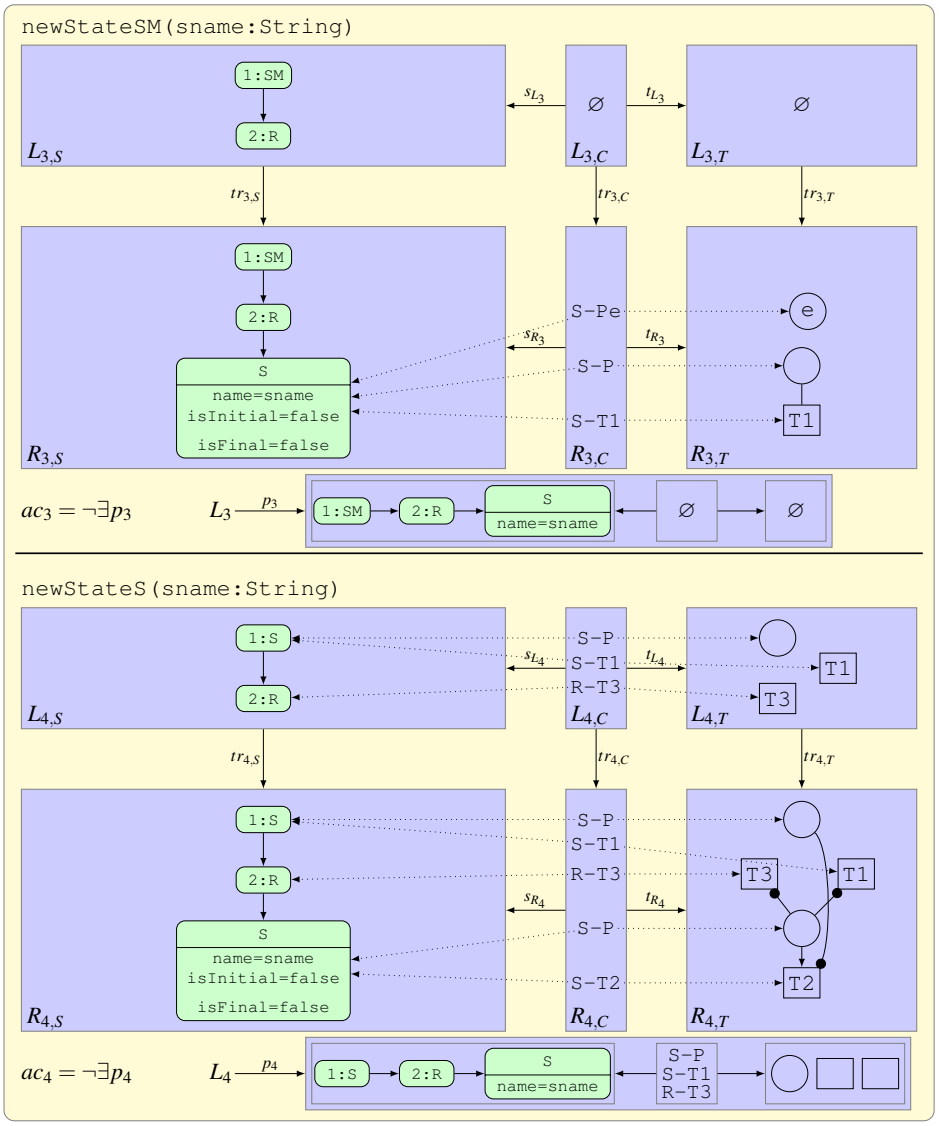


Figure 8: The triple rules `newStateSM` and `newStateS`

that `exit`-events only begin at superstates, i.e. a state containing a region. Note that the objects and morphisms used for the application conditions ac_8, ac_9, and ac_{10} are not shown explicitly, but they correspond to the objects and morphisms used in ac_7.

In Figure 12, the triple rules `newGuard` and `nextGuard` are shown which create the guard conditions of a transition. The guard condition is a state whose corresponding place is connected via a contextual arc to the corresponding net transition. The application conditions ensure that only one guard per transition is allowed and that a transition with `exit`-event is not guarded at all. With the rule `newAction` in Figure 12, an action is added to a transition in the statechart model if none is specified yet.

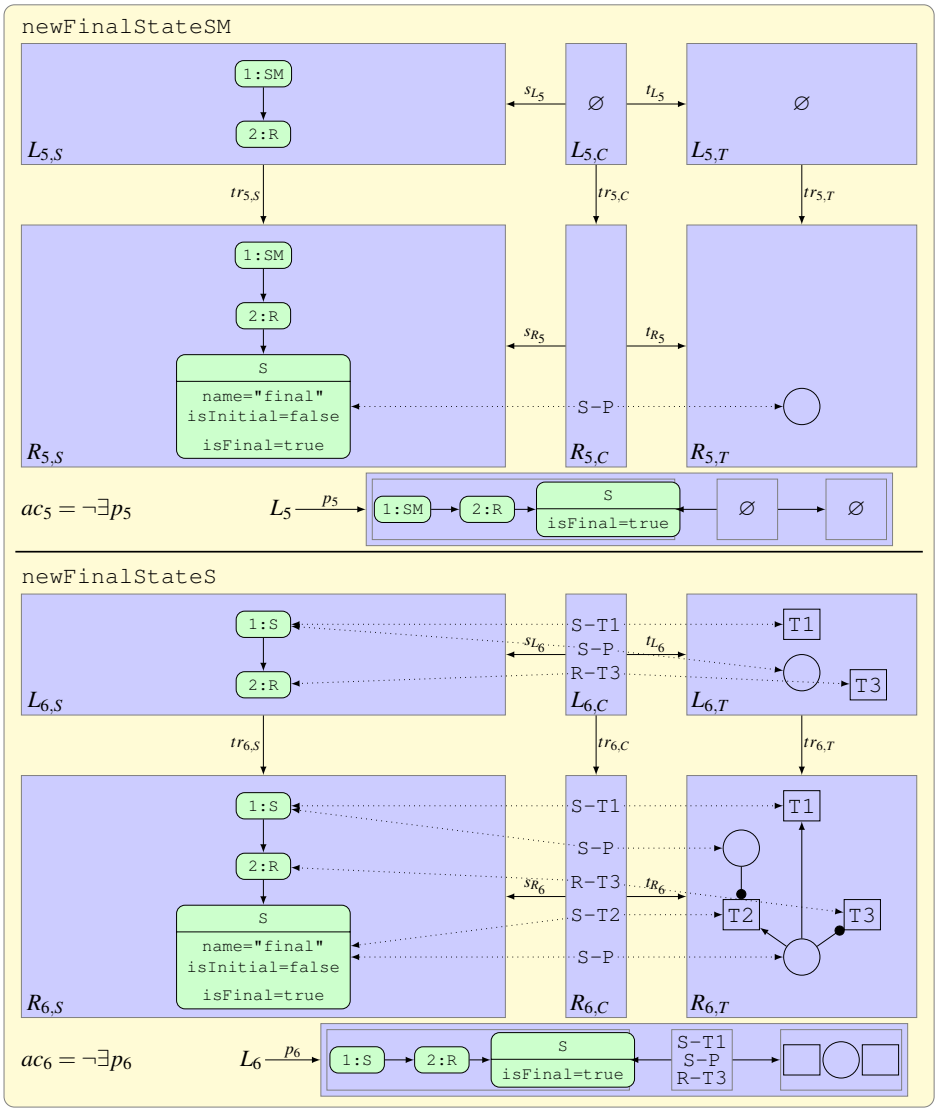


Figure 9: The triple rules `newFinalStateSM` and `newFinalStateS`

An integrated model containing the statechart example in Figure 1 in its source component can be constructed by the application of the following triple rules:

- 1 × `start` creating the state machine,
- 1 × `newRegionSM` creating the one region inside the state machine and the initial state `prod`,
- 1 × `newStateSM` creating the state `error`,
- 4 × `newRegionS` creating one region within `error` including the initial state `call` and the three regions within `prod` including the initial states `produced`, `empty`, and `wait`,
- 4 × `newStateS` creating the state `repair` within `error` and the states `prepare`, `full`, and `consumed` within `prod`,

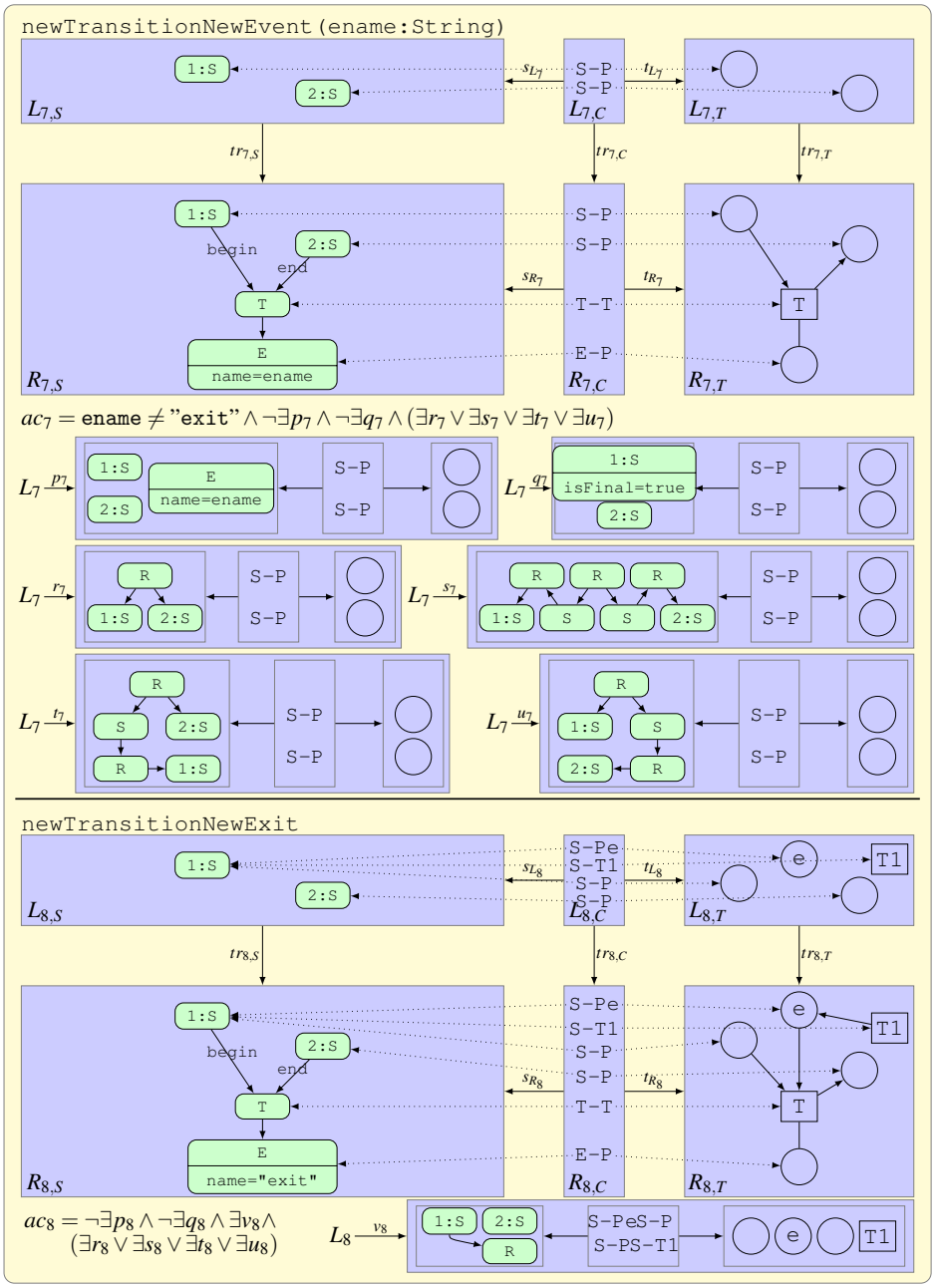


Figure 10: `newTransitionNewEvent` and `newTransitionNewExit`

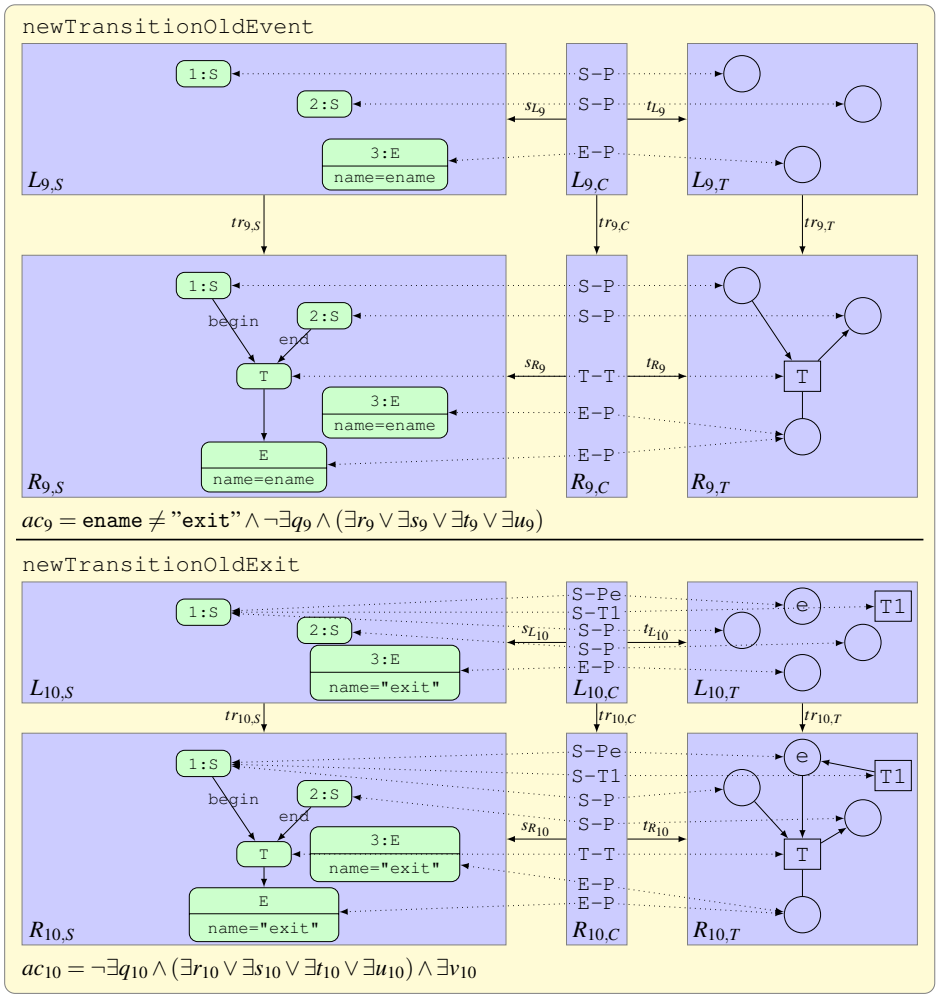


Figure 11: `newTransitionOldEvent` and `NewTransitionOldExit`

- 1 × `newFinalStateSM` creating the final state of the state machine,
- 1 × `newFinalStateS` creating the final state within `error`,
- 9 × `newTransitionNewEvent` creating all transition except for the `exit`-transition between `error` and `prod` and the `next`-transition between `consumed` and `wait`,
- 1 × `newTransitionExit` creating the `exit`-transition between `error` and `prod`,
- 2 × `newTransitionOldEvent` creating the `next`-transition between `consumed` and `wait` with the already known event `next`,
- 2 × `newGuard` creating the guards of the `produce`- and `consume`-transitions,
- 2 × `newAction` creating the actions of the `produce`- and `consume`-transitions.

In the target component we find the Petri net depicted in Figure 13 (without the initial marking), where we have labeled the places and transitions with the names of the corresponding statechart elements and correspondence node names to ease the recognition.

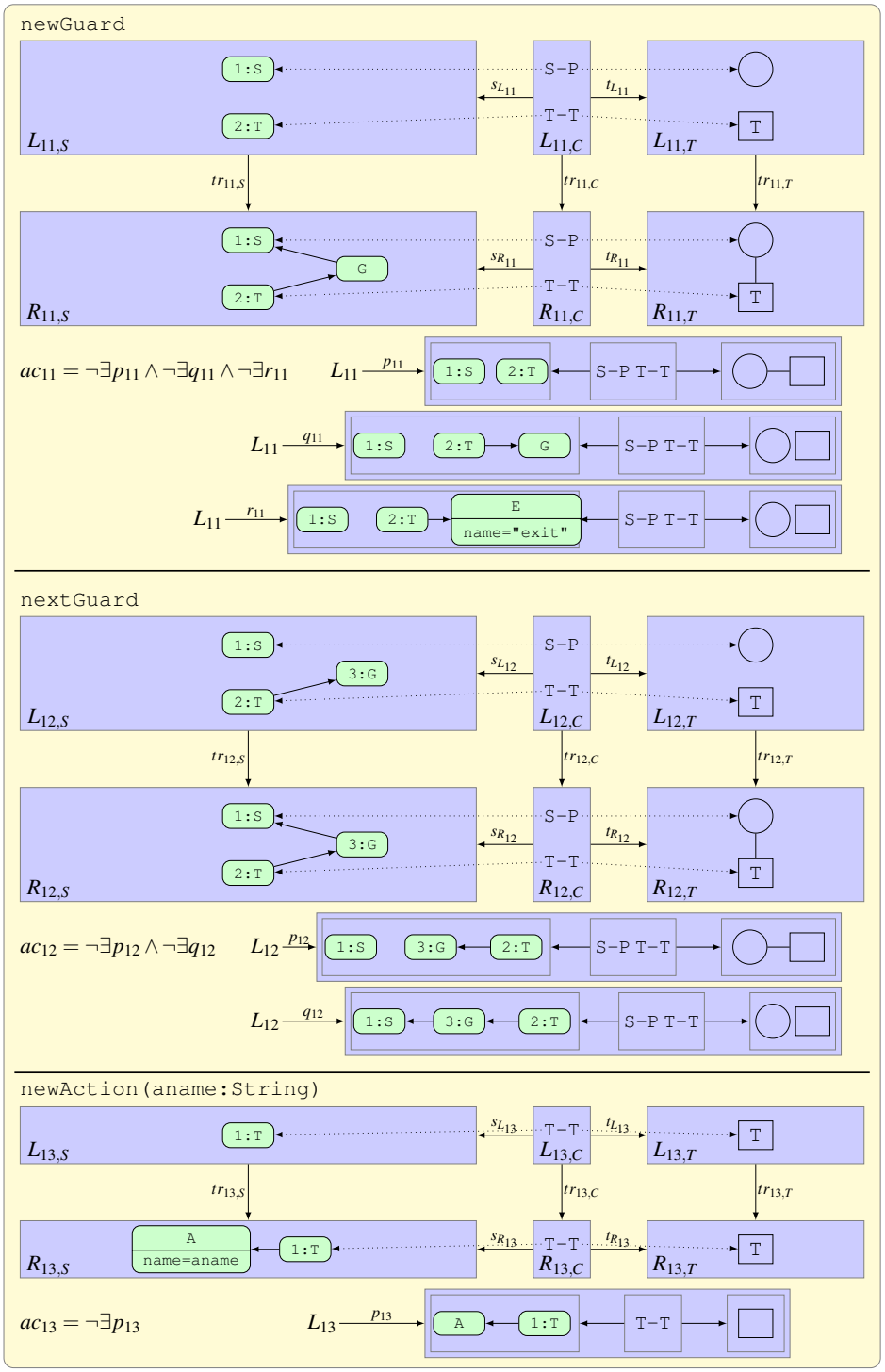


Figure 12: The triple rules `newGuard`, `nextGuard`, and `newAction`

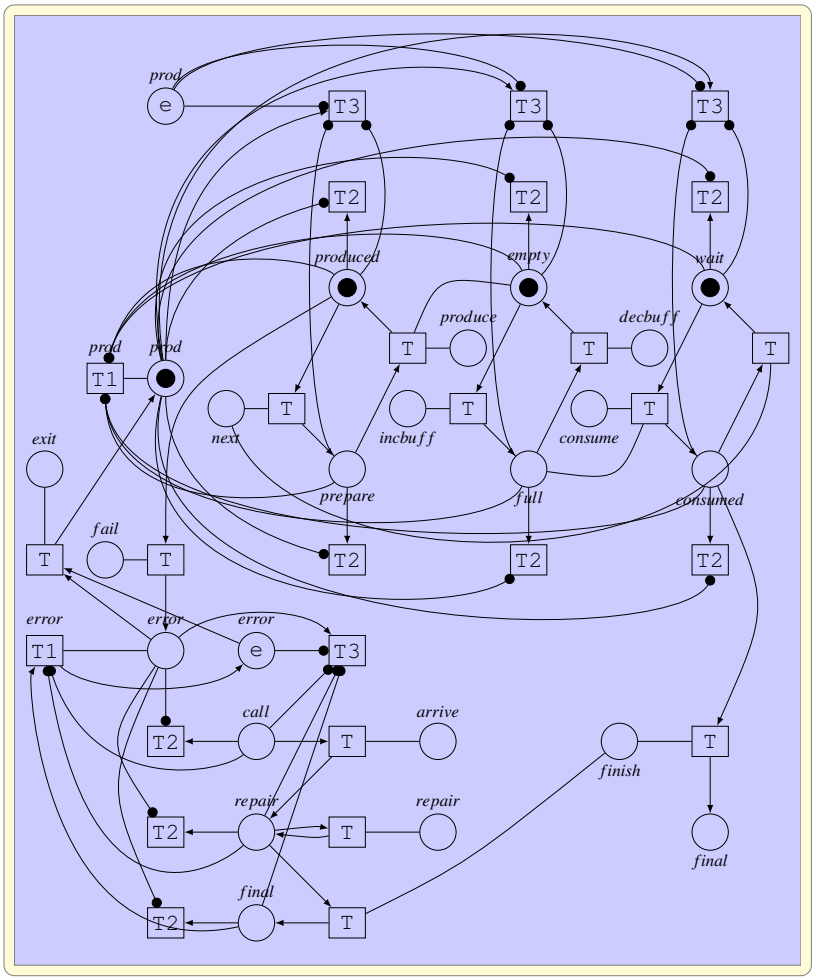


Figure 13: The Petri net corresponding to the statechart in Figure 1

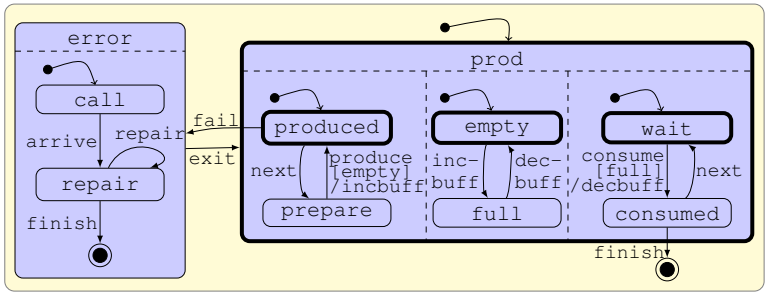


Figure 14: The statechart after the initialization step

We do not want to show the weak simulation relation between the statecharts semantics and the Petri net completely (see [Gol11]), but give some intuition how it works. First, the initialization takes place. For the statechart, this leads to the active states `prod`, `produced`, `empty`, and `wait` as shown in Figure 14 with thicker lines, since the initial state and all its initial substates are invoked. In the Petri nets, the corresponding open places create a token leading to the initial marking depicted in Figure 13.

For the first semantical step, an external trigger element `next` appears. The state transition de-

activates the state `produced` and activates the state `prepare`. In the Petri net, the `next`-place generates a token. Now the `T`-transition with `next` and `produced` as pre-places is activated and fires. Since no other transition is activated, deleting the `next`-token leads to the resulting Petri net simulation step with tokens on the places `prod`, `prepare`, `empty`, and `wait` corresponding to the statechart's current semantical state.

The source rules including suitable derived application conditions represent a generating grammar for our statechart models. All models are typed over the type graph and respect the specified constraints. For the target rules, only a subset of Petri nets can be generated, but all models obtained from transformations using the target rules are well-formed, because they are typed over the Petri net type graph and we cannot generate double arcs. This is due to the fact that the rules either create only arcs from or to a new element or the multiple application is forbidden as for the rule `newGuard` by the expression $\neg\exists p_{11}$ within the application condition ac_{11}.

4 Model Transformations with Application Conditions

As shown by the model transformation from statecharts to Petri nets, rules with application conditions are more expressive and allow to restrict the application of the rules. Thus, we enhance triple rules and combine a triple rule tr without application conditions with an application condition ac over L. Then a triple transformation is applicable if the match m satisfies the application condition ac. From now on, a triple rule denotes a rule with application conditions, while the absence of application conditions is explicitly mentioned.

First, we introduce triple rules which construct the source, connection, and target parts in one step. From these triple rules we derive later the operational source and forward rules for the model transformation.

Definition 1 (Triple rule and transformation) A *triple rule* $tr = (tr : L \to R, ac)$ consists of triple graphs L and R, an $\mathscr{M}$-morphism $tr : L \to R$, and an application condition ac over L.

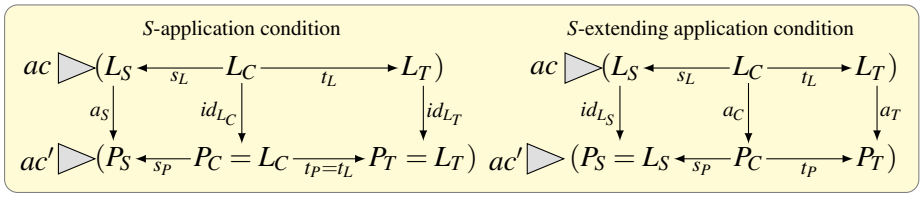
A *direct triple transformation* $G \overset{tr,m}{\Longrightarrow} H$ of a triple graph G via a triple rule tr and a match $m : L \to G$ with $m \models ac$ is given by the direct triple transformation $G \overset{\overline{tr},m}{\Longrightarrow} H$ via the corresponding triple rule without application conditions.

Example 1 *Examples for triple rules using application conditions have been shown in Section 3.*

For the extension of the derived rules with application conditions, we need more specialized application conditions that can be assigned to the source and forward rules.

Definition 2 (Special application conditions) Given a triple rule $tr : L \to R$, an application condition $ac = \exists(a, ac')$ over L with $a : L \to P$ is an

- *S-application condition* if a_C, a_T are identities, i.e. $P_C = L_C$, $P_T = L_T$, and ac' is an S-application condition over P, and
- *S-extending application condition* if a_S is an identity, i.e. $P_S = L_S$, and ac' is an S-extending application condition over P.



Moreover, true is an S- and S-extending application condition, and if ac, ac_i are S- or S-extending application conditions for some index set $\mathscr{I}$ so are $\neg ac$, $\wedge_{i\in\mathscr{I}} ac_i$, and $\vee_{i\in\mathscr{I}} ac_i$.

For the assignment of the application condition ac to the derived rules, the application condition has to be consistent to the source and forward rules, which means that we must be able to decompose ac into S- and S-extending application conditions.

Definition 3 (S-consistent application condition) Given a triple rule $tr = (tr : L \to R, ac)$, then ac is *S-consistent* if it can be decomposed into $ac \cong ac'_S \wedge ac'_F$ such that ac'_S is an S-application condition and ac'_F is an S-extending application condition.

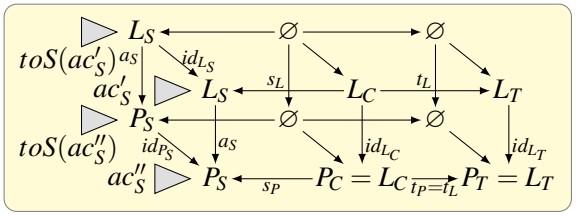
Checking S-consistency for arbitrary application conditions may be complex. Thus, we generally assume that the designer of the triple rules specifies only conjunctions of S-application conditions and S-extending application conditions. From the application point of view, this still provides sufficient expressive power. In fact, the S-application conditions allow for the specification of first order logic (FOL) expressions for the source component and the S-extending ones allow for FOL-expressions on the target component.

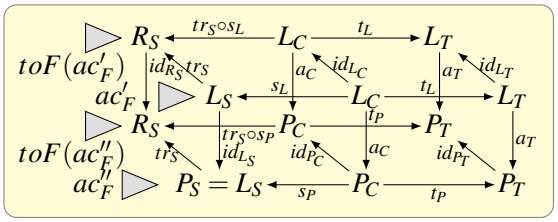
Example 2 *All triple rules in Section 3 have S-consistent application conditions. For example, the application condition ac_7 of the rule* `newTransitionNewEvent` *in Figure 10 is an S-application condition, thus no decomposition is necessary. Moreover, the application condition ac_{11} of the rule* `newGuard` *in Figure 12 can be decomposed into the S-application condition $\neg\exists q_{11} \wedge \neg\exists r_{11}$ and the S-extending application condition $\neg\exists p_{11}$.*

For an S-consistent application condition, we obtain the application conditions of the source and forward rules from the S- and S-extending parts of the application condition, respectively.

Definition 4 (Derived rules with application conditions) Given a triple rule $tr = (tr : L \to R, ac)$ with S-consistent $ac \cong ac'_S \wedge ac'_F$ we translate ac'_S to an application condition $ac_S = toS(ac'_s)$ on $(L_S \leftarrow \varnothing \to \varnothing)$ and ac'_F to an application condition $ac_F = toF(ac'_F)$ on $(R_S \leftarrow L_C \to L_T)$ using the constructions below. This leads to the *source rule* (tr_S, ac_S) and the *forward rule* (tr_F, ac_F).

Given an S-application condition ac'_S and an S-extending application condition ac'_F over L, we define $toS(ac'_S)$ and $toF(ac'_F)$ by





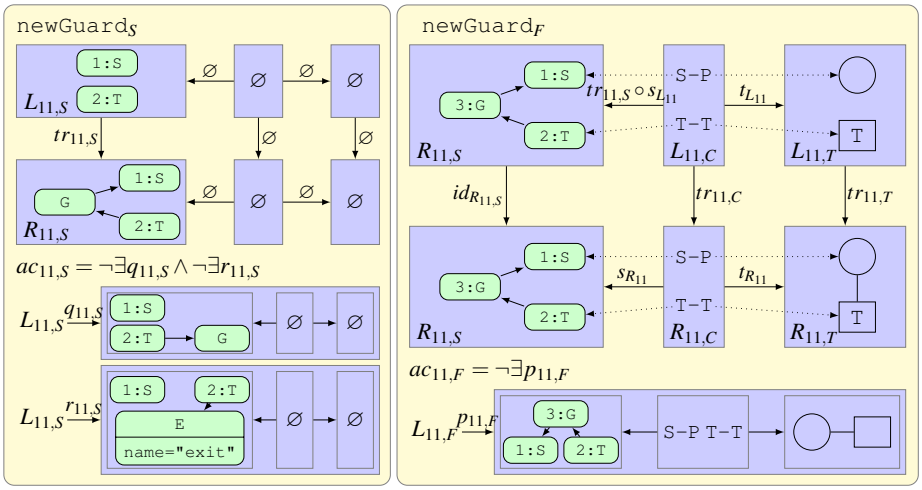


Figure 15: The source and forward rules of newGuard

- $toS(\text{true}) = toF(\text{true}) = \text{true}$,
- $toS(\exists(a, ac''_S)) = \exists((a_S, id_\varnothing, id_\varnothing), toS(ac''_S))$,
- $toF(\exists(a, ac''_F)) = \exists((id_{R_S}, a_C, a_T), toF(ac''_F))$, and
- recursively for composed application conditions.

Example 3 *In Figure 15, the source and forward rules* $\texttt{newGuard}_S$ *and* $\texttt{newGuard}_F$ *of the rule* newGuard *in Figure 12 are shown. The S-application condition* $\neg\exists p_6 \wedge \neg\exists r_6$ *is translated to the source rule, where the source graphs of the original application conditions are kept, but the connection and target graphs are empty now. The S-extending application condition* $\neg\exists q_6$ *is translated to the forward rule, where the source graph is adapted to the new left-hand side.*

Similar to the corresponding result for triple rules without application conditions, in case of S-consistency each triple rule is the E-concurrent rule of its source and forward rules.

Proposition 1 *Given a triple rule* $tr = (tr : L \to R, ac)$ *with S-consistent ac, then* $tr = tr_S *_E tr_F$ *with E being the domain of the forward rule.*

Proof idea. From [EEE+07] we know that this holds for triple rules without application conditions. For the application conditions, this can be shown in two steps using the definition of the application conditions and the shift properties (see [Gol11]).

For the first step, we have to show that $\text{Shift}((id_{L_S}, \varnothing_{L_C}, \varnothing_{L_T}), ac_S) \cong ac'_S$. With $ac_S = toS(ac'_S)$ this is obviously true for $ac'_S = \text{true}$. Consider $ac'_S = \exists(a, ac''_S)$ with $a : L \to P$ and suppose $\text{Shift}((id_{P_S}, \varnothing_{L_S}, \varnothing_{L_C}), toS(ac''_S)) \cong ac''_S$. It follows that $\text{Shift}((id_{L_S}, \varnothing_{L_C}, \varnothing_{L_T}), toS(\exists(a, ac''_S))) \cong \exists(a, \text{Shift}((id_{P_S}, \varnothing_{L_S}, \varnothing_{L_T}), toS(ac''_S)) \cong \exists(a, ac''_S) = ac'_S$ because the shift construction implies that only the trivial squares have to be considered for the index set.

For the second step, we have to show that $\text{L}(e_2, \text{Shift}(id_E, ac_F)) \cong ac'_F$ with $e_2 = (tr_S, id_{L_C}, id_{L_T}) : L \to E$. With $ac_F = toF(ac'_F)$ this is obvious for $ac'_F = \text{true}$. Consider $ac'_F = \exists(a, ac''_F)$ with $\text{L}((L_S \leftarrow P_C \to P_T) \to (R_S \leftarrow P_C \to P_T), \text{Shift}(id, toF(ac''_F))) \cong ac''_F$. Then $(P_S = L_S \overset{s_P}{\leftarrow}$

$P_C \xrightarrow{t_P} P_T$) is the pushout complement constructed for the left-shift-construction and we have that $\mathrm{L}(e_2, \mathrm{Shift}(id_E, toF(\exists(a, ac''_F)))) \cong \mathrm{L}(e_2, \exists((id_{R_S}, a_C, a_T), toF(ac''_F))) \cong \exists((id_{L_S}, a_C, a_T), \mathrm{L}(((L_S \leftarrow P_C \rightarrow P_T) \rightarrow (R_S \leftarrow P_C \rightarrow P_T)), toF(ac''_F)) \cong \exists(a, ac''_F) = ac'_F$. □

Now we want to analyze how a triple transformation can be decomposed into a transformation applying first the source rules followed by the forward rules. Match consistency of the decomposed transformation means that the comatches of the source rules define the source part of the matches of the forward rules. This helps us to define suitable forward model transformations, which have to be source consistent to ensure a valid model. Note, that triple transformation sequences always satisfy the application conditions of the corresponding rules.

Definition 5 (Source and match consistency) Given a sequence $(tr_i)_{i=1,\ldots,n}$ of triple rules with S-consistent application conditions leading to corresponding sequences $(tr_{iS})_{i=1,\ldots,n}$ and $(tr_{iF})_{i=1,\ldots,n}$ of source and forward rules. A triple transformation sequence $G_{00} \xRightarrow{tr_S^*} G_{n0} \xRightarrow{tr_F^*} G_{nn}$ via first $tr_{1S}, \ldots, tr_{nS}$ and then $tr_{1F}, \ldots, tr_{nF}$ with matches m_{iS} and m_{iF} and comatches n_{iS} and n_{iF}, respectively, is *match consistent* if the source component of the match m_{iF} is uniquely defined by the comatch n_{iS}.

A triple transformation $G_{n0} \xRightarrow{tr_F^*} G_{nn}$ is called *source consistent* if there is a match consistent sequence $G_{00} \xRightarrow{tr_S^*} G_{n0} \xRightarrow{tr_F^*} G_{nn}$.

We can split a transformation $G_0 \xRightarrow{tr_1} G_1 \Rightarrow \ldots \xRightarrow{tr_n} G_n$ into transformations $G_0 \xRightarrow{tr_{1S}} G'_0 \xRightarrow{tr_{1F}} G_1 \Rightarrow \ldots \xRightarrow{tr_{nS}} G'_{n-1} \xRightarrow{tr_{nF}} G_n$. But to apply first the source and then the forward rules, these have to be independent in a certain sense. In the following theorem, we show that such a decomposition into a match consistent transformation can be found and, vice versa, each match consistent transformation can be composed to a transformation via the corresponding triple rules if the application conditions are S-consistent. This result is an extension of the corresponding result for triple transformations without application conditions [EEE+07] and with negative application conditions [EHS09]. It is essential for concepts and results of model transformations with application conditions below.

Theorem 1 (Decomposition and composition) *For triple transformation sequences with S-consistent application conditions the following holds:*

1. ***Decomposition:*** *For each triple transformation sequence $G_0 \xRightarrow{tr_1} G_1 \Rightarrow \ldots \xRightarrow{tr_n} G_n$ there is a corresponding match consistent triple transformation sequence $G_0 = G_{00} \xRightarrow{tr_{1S}} G_{10} \Rightarrow \ldots \xRightarrow{tr_{nS}} G_{n0} \xRightarrow{tr_{1F}} G_{n1} \Rightarrow \ldots \xRightarrow{tr_{nF}} G_{nn} = G_n$.*
2. ***Composition:*** *For each match consistent triple transformation sequence $G_{00} \xRightarrow{tr_{1S}} G_{10} \Rightarrow \ldots \xRightarrow{tr_{nS}} G_{n0} \xRightarrow{tr_{1F}} G_{n1} \Rightarrow \ldots \xRightarrow{tr_{nF}} G_{nn}$ there is a triple transformation sequence $G_{00} = G_0 \xRightarrow{tr_1} G_1 \Rightarrow \ldots \xRightarrow{tr_n} G_n = G_{nn}$.*
3. ***Bijective Correspondence:*** *Composition and Decomposition are inverse to each other.*

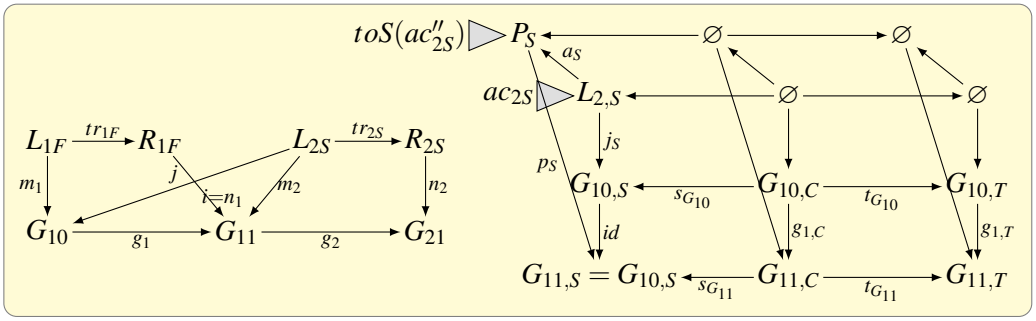
Proof idea. This result has been shown in [EEE+07] for triple rules without application conditions. We use the facts that $tr_i = tr_{iS} *_{E_i} tr_{iF}$, as shown in Prop. 1, and that the transforma-

tions via tr_{iS} and tr_{jF} are sequentially independent for $i > j$. This is shown in [EEE+07] for rules without application conditions and can be extended to triple rules with application conditions as shown in the following. Thus, the proof from [EEE+07] can be done analogously for rules with application conditions. The main idea of the proof is that a triple transformation sequence $G_0 \overset{tr_1}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_n}{\Longrightarrow} G_n$ can be decomposed into a transformation sequence $G_0 \overset{tr_{1S}}{\Longrightarrow} G'_1 \overset{tr_{1F}}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G'_n \overset{tr_{nF}}{\Longrightarrow} G_n$. The sequential independence of tr_{iS} and tr_{jF} for $i > j$ allows us to shift all source rules to the beginning and all forward rules to the end of the sequence leading to an equivalent transformation sequence $G_0 = G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G_{n0} \overset{tr_{1F}}{\Longrightarrow} G_{n1} \Rightarrow \ldots \overset{tr_{nF}}{\Longrightarrow} G_{nn} = G_n$.

It suffices to show that the transformations $G_{10} \overset{tr_{1F},m_1}{\Longrightarrow} G_{11} \overset{tr_{2S},m_2}{\Longrightarrow} G_{21}$ are sequentially independent. From the sequential independence without application conditions we obtain morphisms $i : R_{1F} \to G_{11}$ with $i = n_1$ and $j : L_{2S} \to G_{10}$ with $g_1 \circ j = m_2$.

It remains to show the compatibility with the application conditions:

- $j \models ac_{2S}$: $ac_{2S} = toS(ac'_{2S})$, where ac'_{2S} is an S-application condition. For $ac'_{2S} = \text{true}$, also $ac_{2S} = \text{true}$ and therefore $j \models ac_{2S}$. Suppose $ac'_{2S} = \exists(a, ac''_{2S})$ leading to $ac_{2S} = \exists((a_S, id_\varnothing, id_\varnothing), toS(ac''_{2S}))$. Moreover, tr_{1F} is a forward rule, i. e. it does not change the source component and $G_{11,S} = G_{10,S}$.

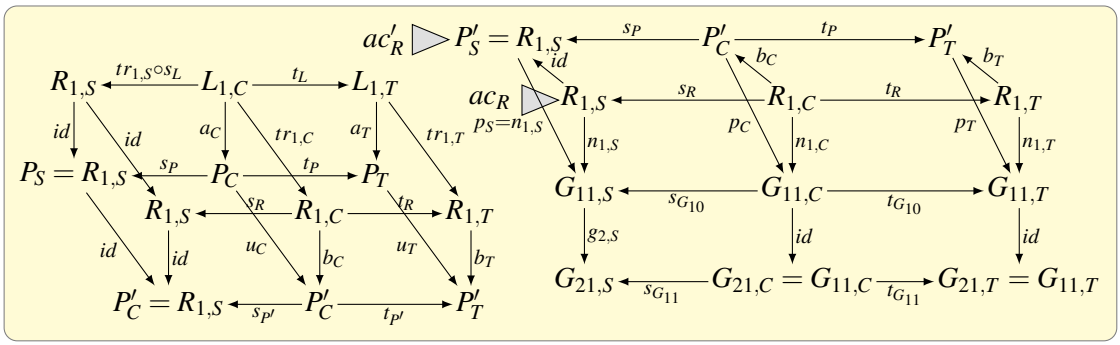


  We know that $m_2 = g_1 \circ j \models ac_{2S}$, which means that there exists $p : P \to G_{11}$ with $p \circ a = g_1 \circ j$, $p \models toS(ac''_{2S})$, and $p_C = \varnothing$, $p_T = \varnothing$. Then there exists $q : P \to G_{10}$ with $q = (p_S, \varnothing, \varnothing)$, $q \circ a = (p_S \circ a_S, \varnothing, \varnothing) = j$, and $q \models toS(ac''_{2S})$ because all objects occuring in $toS(ac''_{2S})$ have empty connection and target components. This means that $j \models ac_{2S}$ for this case, and can be shown recursively for composed ac_{2S}.

- $g_2 \circ n_1 \models ac_R := \mathrm{R}(tr_{1F}, ac_{1F})$: $ac_{1F} = toF(ac'_{1F})$, where ac'_{1F} is an S-extending application condition. For $ac'_{1F} = \text{true}$ also $ac_{1F} = \text{true}$ and $ac_R = \text{true}$, therefore $g_2 \circ n_1 \models ac_R$. Now suppose $ac'_{1F} = \exists(a, ac''_{1F})$ leading to $ac_{1F} = \exists((id_{R_{1,S}}, a_C, a_T), toF(ac''_{1F}))$ and $ac_R = \exists((id_{R_{1_S}}, b_C, b_T), ac'_R)$ by component-wise pushout construction for the right-shift with $ac'_R = \mathrm{R}(u, toF(ac''_{1F}))$. Moreover, tr_{2S} is a source rule which means that $g_{2,C}$ and $g_{2,T}$ are identities.

  From the shift property of application conditions we know that $n_1 \models ac_R$ using that $m_1 \models ac_{1F}$. This means that there is a morphism $p : P \to G_{11}$ with $p \circ a = n_1$, $p \models ac'_R$, and $p_S = n_{1,S}$. It follows that $g_2 \circ p \circ a = g_2 \circ n_1$ and $g_2 \circ p = (g_{2,S} \circ p_S, p_C, p_T) \models ac'_R$, because it only differs from p in the S-component, which is identical in all objects occuring in ac'_R.

This means that $g_2 \circ n_1 \models ac_R = \exists(a, ac'_R)$, and can be shown recursively for composed ac_R.



□

Based on source consistent forward transformations we define model transformations, where we assume that the start graph is the empty graph.

Definition 6 (Model transformation) A *(forward) model transformation sequence* $(G_S, G_0 \overset{tr^*_F}{\Longrightarrow} G_n, G_T)$ is given by a source graph G_S, a target graph G_T, and a source consistent forward transformation $G_0 \overset{tr^*_F}{\Longrightarrow} G_n$ with $G_0 = (G_S \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} \varnothing)$ and $G_{n,T} = G_T$.

A *(forward) model transformation* $MT_F : VL_S \Rrightarrow VL_T$ is defined by all (forward) model transformation sequences.

Definition 7 (Model transformation SC2PN) For our triple transformations, the triple rules are given by the set $TR = \{$`start`, `newRegionSM`, `newRegionS`, `newStateSM`, `newStateS`, `newFinalStateSM`, `newFinalStateS`, `newTransitionNewEvent`, `newTransitionNewExit`, `newTransitionOldEvent`, `newTransitionOldExit`, `newGuard`, `nextGuard`, `newAction`, `newTriggerElement`$\}$ as introduced in Section 3.

The model transformation SC2PN from statecharts to Petri nets is defined by all forward model transformations using the forward rules TR_F.

The source rules represent a generating grammar for our statechart models. Moreover, the restriction of all derived triple graphs to their source part, the language constructed by the source rules, and the statechart language VL_{SC} are equal.

Proposition 2 (Comparison of statechart languages) *Consider the languages* $VL_S = \{G_S \mid \exists$ *triple transformation* $\varnothing \overset{\texttt{start}}{\Longrightarrow} \overset{tr^*}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T)$ *via rules in* $TR\}$, $VL_{S0} = \{G_S \mid \exists$ *triple transformation* $\varnothing \overset{\texttt{start}_S}{\Longrightarrow} \overset{tr^*_S}{\Longrightarrow} (G_S \leftarrow \varnothing \rightarrow \varnothing)$ *via source rules in* $TR_S\}$, *and* VL_{SC} *as defined by the type graph and constraints. Then we have that* $VL_S = VL_{S0} = VL_{SC}$.

Proof idea. $VL_S \subseteq VL_{S0}$: For a statechart $G_S \in VL_S$ there is a transformation $\varnothing \overset{\texttt{start}}{\Longrightarrow} \overset{tr^*}{\Longrightarrow} (G_S \leftarrow G_C \rightarrow G_T) = G_n$, which can be decomposed with Theorem 1 into a corresponding sequence $\varnothing \overset{start_S}{\Longrightarrow} \overset{tr^*_S}{\Longrightarrow} (G_S \leftarrow \varnothing \rightarrow \varnothing) \overset{start_F}{\Longrightarrow} \overset{tr^*_F}{\Longrightarrow} G_n$. This means that $G_S \in VL_{S0}$.

$VL_{S0} \subseteq VL_{SC}$: For a statechart $G_S \in VL_{S0}$ there is a transformation $\varnothing \overset{start_S}{\Longrightarrow} \overset{tr_S^*}{\Longrightarrow} (G_S \leftarrow \varnothing \rightarrow \varnothing)$. G_S is typed over the type graph TG_S and respects all the specified constraints. This means that $G_S \in VL_{SC}$.

$VL_{SC} \subseteq VL_S$: Given a statechart model $M \in VL_{SC}$ we have to show that we find a transformation sequence $\varnothing \overset{\mathtt{start}}{\Longrightarrow} \overset{tr^*}{\Longrightarrow} G$ with $G_S = M$. We can show this by arguing about the composition of M and how to select the corresponding triple rule creating each element in M in the source part. This means that $M \in VL_S$. □

Example 4 *As explained for our example transformation in Section 3, applying the corresponding source rule sequence to the empty start graph we obtain our statechart example. This statechart model can be transformed into the Petri net via the forward rules. This triple transformation is source consistent, since the matches of the source parts for the forward rules are uniquely defined by the comatches of the source rules. Thus, we actually obtain a model transformation sequence from the statechart model in Figure 1 to the Petri net in Figure 13.*

For all notions and results concerning source and forward rules, we obtain the dual notions and results for target and backward rules. Thus, an application condition ac is T-consistent if it can be decomposed into $ac \cong ac'_T \wedge ac'_B$, where ac'_T is a T-application condition with identities a_S, a_C and ac'_B is a T-extending application condition with identity a_T. This leads to target and backward rules with application conditions and the dual composition and decomposition properties for triple transformation sequences with T-consistent application conditions. Moreover, a backward model transformation sequence $(G_T, G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n, G_S)$ is based on a target consistent backward transformation $G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n$ with $G'_0 = (\varnothing \overset{\varnothing}{\longleftarrow} \varnothing \overset{\varnothing}{\longrightarrow} G_T)$ and $G'_{n,S} = G_S$.

4.1 Results for Model Transformations with Application Conditions

Based on Theorem 1 we can show correctness, completeness, backward information preservation, and termination of model transformations. The first result shows that transformations are correct and complete regarding the source and target languages.

Theorem 2 (Correctness and completeness w.r.t. VL_S, VL_T) *Each model transformation sequence $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G_n, G_T)$ and $(G_T, G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n, G_S)$ is correct with respect to the source and target languages, i.e. $G_S \in VL_S$ and $G_T \in VL_T$.*

For each $G_S \in VL_S$ there is a corresponding $G_T \in VL_T$ such that there is a model transformation sequence $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G_n, G_T)$. Similarly, for each $G_T \in VL_T$ there is a corresponding $G_S \in VL_S$ such that there is a model transformation sequence $(G_T, G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n, G_S)$.

Proof. If $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ is source consistent we have a match consistent sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ by Definition 5 . By composition in Theorem 1 there is a triple transformation $\varnothing \overset{tr^*}{\Longrightarrow} G_n$ with $G_S = G_{n,S} \in VL_S$ and $G_T \in VL_T$.

For $G_S \in VL_S$ there exists a triple transformation $\varnothing \overset{tr^*}{\Longrightarrow} G$, which can be decomposed by Theorem 1 into a match consistent sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G_0 = (G_S \overset{\varnothing}{\longleftarrow} \varnothing \overset{\varnothing}{\longrightarrow} \varnothing) \overset{tr_F^*}{\Longrightarrow} G$, and by

definition $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G, G_T)$ is the required model transformation sequence with $G_T \in VL_T$.

Dually, this holds for backward model transformation sequences. □

Example 5 *Since our example in Section 3 represents a well-defined model transformation sequence, our statechart and Petri net are correct. Moreover, for each valid statechart model we obtain a correct Petri net model, and vice versa. Note, that for the backward translation this only holds for Petri nets which are correct w.r.t. our target language, and not the language of all well-formed Petri nets.*

A forward model transformation from G_S to G_T is backward information preserving concerning the source component if there is a backward transformation sequence from G_T leading to the same source graph G_S.

Definition 8 (Backward information preserving) A forward transformation sequence $G \overset{tr_F^*}{\Longrightarrow} H$ is *backward information preserving* if for the triple graph $H' = (\varnothing \overset{\varnothing}{\longleftarrow} \varnothing \overset{\varnothing}{\longrightarrow} H_T)$ there is a backward transformation sequence $H' \overset{tr_B^*}{\Longrightarrow} G'$ with $G'_S \overset{\sim}{=} G_S$.

This theorem is an extension of the corresponding result in [EEE+07] to triple transformations with application conditions.

Theorem 3 (Backward information preservation) *If all triple rules are S- and T-consistent, a forward transformation $G \overset{tr_F^*}{\Longrightarrow} H$ is backward information preserving if it is source consistent.*

Proof. If $G \overset{tr_F^*}{\Longrightarrow} H$ is a source consistent sequence then by Def. 5 there exists a match consistent sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G \overset{tr_F^*}{\Longrightarrow} H$ leading to the triple transformation sequence $\varnothing \overset{tr^*}{\Longrightarrow} H$ using Theorem 1. From the decomposition, we also obtain a match consistent sequence $\varnothing \overset{tr_T^*}{\Longrightarrow} H' \overset{tr_B^*}{\Longrightarrow} H$ using the target and backward rules, with $H'_T = H_T$ and $H'_C = H'_S = \varnothing$. Thus, $G \overset{tr_F^*}{\Longrightarrow} H$ is backward information preserving. □

Example 6 *The Petri net in Figure 13 can be transformed into the statechart in Figure 1 using the backward rules of our model transformation in the same order as the forward rules were used for the forward transformation. Indeed, this holds for each Petri net obtained of a model transformation sequence from a valid statechart model.*

If the source and target rules are creating, i.e. each rule actually creates at least one element, forward and backward transformation sequences are terminating. This means that we do not find infinite model transformation sequences.

Theorem 4 (Termination) *Consider a source model $G_S \in VL_S$ (target model $G_T \in VL_T$) and a set of triple rules such that G_S (G_T) and all rule components are finite on the graph part and the triple rules are creating on the source (target) component. Then each model transformation sequence $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G_n, G_T)$ $((G_T, G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n, G_S))$ is terminating, i.e. any extended sequence $G_0 \overset{tr_F^*}{\Longrightarrow} G_n \overset{tr'^+_F}{\Longrightarrow} G_m$ $(G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n \overset{tr'^+_B}{\Longrightarrow} G'_m)$ is not source (target) consistent.*

Proof. Let $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ be a source consistent forward sequence such that $\varnothing \overset{tr_S^*}{\Longrightarrow} G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ is match consistent, i.e. each comatch $n_{i,S}$ determines the source component of the match $m_{i,F}$. Thus, also each forward match $m_{i,F}$ determines the corresponding comatch $n_{i,S}$. By uniqueness of pushout complements along $\mathscr{M}$-morphisms the comatch $n_{i,S}$ determines the match $m_{i,S}$ of the source step, thus $m_{i,F}$ determines $m_{i,S}$ $(*)$.

If $G_0 \overset{tr_F^*}{\Longrightarrow} G_n \overset{tr_{(n+1,F)},m_{(n+1,F)}}{\Longrightarrow} G_{n+1} \overset{tr''^*_F}{\Longrightarrow} G_m$ is a source consistent forward sequence then there is a corresponding source sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G' \overset{tr_{n+1,S}}{\Longrightarrow} G'' \overset{tr''^*_S}{\Longrightarrow} G_0$ leading to match consistency of the complete sequence $\varnothing \Rightarrow^* G_m$. Using $(*)$ it follows that $G' \cong G_0$, which implies that we have a transformation step $G_0 \overset{tr_{n+1,S}}{\Longrightarrow} G'' \subseteq G_0$, because triple rules are non-deleting. This is a contradiction to the precondition that each rule is creating on the source component implying that $G' \ncong G_0$. Therefore, the forward transformation sequence $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ cannot be extended and is terminating.

Dually, this can be shown for backward model transformation sequences. □

Example 7 *All triple rules in our example in Section 3 are finite on the graph part and source creating. Thus, all model transformation sequences based on finite statechart models are terminating. Note, that this does not hold for the backward direction, since the rule* `newAction` *is not target creating. Thus, the corresponding backward rule can be applied infinitary often.*

5 Conclusion

In this paper, we have extended the theory of model transformations based on TGGs to rules with nested application conditions [HP09], which are known to be equivalent to first order logic (FOL) on graphs. Using the slight restriction to S-consistent application conditions we have shown that the main results known for model transformations are preserved. In fact, this is a substantial extension of the existing theory, because S-consistent application conditions provide the expressive power of FOL separately for the source and target components of triple graphs, respectively. This enhances the expressiveness of model transformations including that of the generation of source and/or target languages. We have discussed in detail a model transformation from statecharts to Petri nets, where the use of application conditions allows to specify and translate more general statecharts then those considered in [EEPT06]. There, an inplace model transformation is used, which means that the model itself is changed in contrast to our approach, where the original source model is kept and an additional target model is created. We have presented main results for termination, correctness, completeness, and information preservation extending those for the case with NACs in [EHS09] and without NACs in [EEE+07].

Our new results are based on the Local Church–Rosser, Parallelism, and Concurrency Theorems with nested application conditions in [EHL10]. As future work it remains to extend also the results concerning functional behaviour in [HEOG10] and [HEGO10] to the case of rules with nested application conditions based on the "on-the-fly construction" in [EEHP09]. This would allow to meet the "Grand Research Challenge of the TGG Community" in [SK08] for our enhanced framework.

It is out of the scope of this paper to show that our model transformation from statecharts to

Petri nets is semantically correct, where the semantics of the source and target language could be based on a suitable operational semantics. For statecharts, an operational semantics based on amalgamated graph transformation is presented in [GBEE11]. In [Gol11], also an operational semantics for Petri nets using amalgamated graph transformation is defined and the model transformation given in this paper is shown to be semantics-preserving. It is future work to obtain general criteria for semantical correctness of model transformations.

Another future point of work is the construction of source and forward application conditions for general, not necessarily *s*-consistent application conditions. Obviously, in this case a different property for the compatibility of the source and forward solutions would be required to ensure the corresponding decomposition and composition result.

Bibliography

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *Proceedings of FASE 2007*. LNCS 4422, pp. 72–86. Springer, 2007.

[EEH08] H. Ehrig, C. Ermel, F. Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In Karsai and Taentzer (eds.), *Proceedings of GraMoT 2008*. Pp. 9–16. ACM, 2008.

[EEHP09] H. Ehrig, C. Ermel, F. Hermann, U. Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars. In Schürr and Selic (eds.), *Proceedings of MODELS 2009*. LNCS 5795, pp. 241–255. Springer, 2009.

[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.

[EHS09] H. Ehrig, F. Hermann, C. Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *ECEASST* 18:1–18, 2009.

[GBEE11] U. Golas, E. Biermann, H. Ehrig, C. Ermel. A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. *ECEASST* 39:1–24, 2011. This volume.

[GL06a] E. Guerra, J. de Lara. Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. Technical report UC3M-TR-CS-2006-00, Universidad Carlos III, Madrid, Spain, 2006.

[GL06b] E. Guerra, J. de Lara. Model View Management with Triple Graph Grammars. In Corradini et al. (eds.), *Proceedings of ICGT 2006*. LNCS 4178, pp. 351–366. Springer, 2006.

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

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Bézivin et al. (eds.), *Proceedings of MDI 2010*. Pp. 22–31. ACM, 2010.

[HEOG10] F. Hermann, H. Ehrig, F. Orejas, U. Golas. Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In *Proceedings of ICGT 2010*. LNCS 6372, pp. 155–170. Springer, 2010.

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

[KS06] A. König, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *ENTCS* 148(1):113–150, 2006.

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

[Pet80] C. Petri. Introduction to General Net Theory. In Brauer (ed.), *Net Theory and Applications*. LNCS 84, pp. 1–19. Springer, 1980.

[Sch94] A. Schürr. Specification of Graph Translators With Triple Graph Grammars. In Tinhofer (ed.), *Proceedings of WG 1994*. LNCS 903, pp. 151–163. Springer, 1994.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In Ehrig et al. (eds.), *Proceedings of ICGT 2008*. LNCS, pp. 411–425. Springer, 2008.

[TEG+05] G. Taentzer, K. Ehrig, E. Guerra, J. Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varró, S. Varró-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proceedings of MTP 2005*. 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/.`