Explicit Modeling of Semantics Associated with Composite States in UML Statecharts[1]

Zhaoxia Hu and Sol M. Shatz

Concurrent Software Systems Laboratory
Department of Computer Science
University of Illinois at Chicago
{zhu, shatz}@cs.uic.edu

Abstract: UML statecharts are used for describing dynamic aspects of system behavior. The work presented here extends a general Petri net-based methodology to support formal modeling of UML statecharts. The approach focuses on the specific task of generating explicit transition models associated with the hierarchical structure of statechart. We introduce a state-transition notation to serve as an intermediate model for conversion of UML statecharts, and in particular, the complexity of composite states, to other target specifications. By defining a process for deriving, from UML statecharts, a state-transition notation that can serve as an intermediate state machine model, we seek to deepen understanding of modeling practices and help bridge the gap between model development and model analysis. This work covers all of the primary issues associated with the hierarchical structure of composite states, including entry and exit transitions, transition priorities, history states, and event dispatching. Thus, the results provide an important step forward toward the goal of modeling increasingly complex semantics of UML statecharts.

Keywords: Composite States, Petri Nets, Statecharts, State-transition Notation, UML

1 Introduction

The Object Management Group (OMG) adopted a new paradigm for software development called Model Driven Architecture (MDA) [Poo01] to recognize the fact that models are important artifacts of software development and they serve as a basis for systems as they evolve from requirements through implementation. In MDA, models are defined in the Unified Modeling Language (UML), which is a graphical language for visualizing, specifying, constructing, and documenting a software-intensive system [BJR99]. MDA raises the importance of addressing many primary issues in the current standard of UML. One of the issues, the semantics of UML, is the subject of active discussion and much research activity [Rum98]. UML is a semi-formal language, since some parts of it are specified formally, while other parts,

[1] This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-01-1-1-0672, and the U.S. National Science Foundation under grant number CCR-9988168.

for instance dynamic semantics, are defined informally [OMG01]. The lack of formal dynamic semantics for this language limits its capability for analyzing defined specifications. The need for a formal semantics for UML is motivated in the literature [BL+00, FE+98], and the pUML (precise UML) group has been created to achieve this goal [PUML]. A number of projects discuss formalizing UML by mapping the UML notation to an alternative notation to give the UML notation a precise semantics and achieve UML verification. This paper is concerned with one core component of UML – statechart diagrams [OMG03].

Statechart diagrams are used for describing dynamic aspects of system behavior in the framework of UML. One line of research aims to give a formal semantics to the statecharts in UML [Bee01, BP01, CHS00, GP98, Kus01, LMM99, LP99, and Reg02]. Another related set of work has been carried out in the area of modeling to support validation and analysis of UML statecharts [BC+, GI00, GL+00, MC99, PG00, and PL99]. The work in this paper relates most closely to the second line of research. Our earlier work, [SSH01, HS], provided a general approach for mapping UML statechart models to colored Petri nets (CPNs) [KCJ98] for the purpose of allowing the use of existing net analysis techniques. We will give an overview of this approach in Section 2. The new work presented here extends the general approach by focusing on the specific task of making explicit the semantics of composite states, including handling the interrelated features of composite states: entry transitions, exit transitions, transition priority and history states, and event dispatching. This work defines an important step forward toward the goal of modeling increasingly complex semantics of UML statecharts.

1.1 Statecharts and Colored Petri Nets

We now provide a quick introduction to statecharts. In UML, statechart diagrams describe the dynamic behavior of the system. UML statecharts are an object-based variant of classical (Harel) statecharts [Har87]. Note that we use the simple form "statecharts" to refer to the UML statecharts in the rest of this paper. We use the UML standard defined in [OMG03] supplemented with the description presented in [BJR99].

Statecharts extend finite state machines with composite states to facilitate describing highly complex behaviors. This extension includes hierarchical structure of states and concurrency. The hierarchical structure of states is implemented with a special type of state, *composite state*. The concurrency feature is handled by one type of composite state, a *concurrent composite state*. Overall, a state of a statechart can be one of the following:

- A *simple state*, which has no substructure.
- A *sequential composite state*, containing sequential substates. If the sequential composite state is active, exactly one of its substates is also active.

- A *concurrent composite state*, containing concurrent substates. If the concurrent composite state is active, one nested state from each one of the concurrent substates is also active.

The execution semantics of a state machine [OMG03] are described in terms of a hypothetical machine whose key components are:

- An event queue that holds incoming event instances until they are dispatched.
- An event dispatcher mechanism that selects and de-queues event instances from the event queue for processing.
- An event processor that processes dispatched event instances.

Petri nets are a mathematically precise model, and so both the structure and the behavior of Petri net models can be described using mathematical concepts. We assume that the reader has some familiarity with basic Petri net modeling [Mur89], but we can start with a general reminder of Petri net concepts. By mathematical definition, a Petri net is a bipartite, directed graph consisting of a set of nodes and a set of arcs, supplemented with a distribution of tokens in places. A bipartite graph is a graph with a set of two types of nodes and no arc connecting two nodes of the same type. For (ordinary) Petri nets, we use the following three-tuple definition:

$PN = (T, P, A)$,

where $T = \{t_1, t_2, \ldots, t_n\}$, a set of nodes called transitions,

$P = \{p_1, p_2, \ldots, p_m\}$, a set of nodes called places,

$A \subseteq (T \times P) \cup (P \times T)$, a set of directed arcs.

Note that an arc connects a transition to a place or a place to a transition. The distribution of tokens among places at certain time defines the current state of the modeled system. Transitions are enabled to fire when certain conditions are satisfied, resulting in a change of token distribution for places. With its formal representation and well-defined syntax and semantics, Petri nets can be "executed" to perform model analysis and verification.

Colored Petri nets (CPNs) are one type of Petri net. In colored Petri nets, tokens are differentiated by *colors*, which are data types. Places are typed by *colorsets*, which specify which type of tokens can be deposited into a certain place. Arcs are associated with inscriptions, which are expressions defined with data values, variables, and functions. Arc inscriptions are used to specify the enabling condition of the associated transition as well as the tokens that are to be generated by the transition.

1.2 Scope of Work

UML statecharts have a large number of complex and interrelated features. This paper does not attempt to cover all of these features, but focuses on concepts that are uniquely associated with composite states.

To help define the scope of our work, let's consider three categories of statecharts. Category I are the most basic statecharts, which consist of simple states, simple transitions, signal events, actions that generate events, and initial states. Category I statecharts are the subject of our previous work [SSH01, HS], which will be described in Section 2. Category II statecharts include also the important feature of composite states, which have broad implications. For example, Category II statecharts incorporate entry transitions, exit transitions, completion transitions, final states, and history states. Modeling the features of Category II statecharts is the subject of this paper. Category III includes other statechart features, such as guards, deferred events, activities, non-signal events, and actions that involve variables. These features have not yet been fully evaluated, but with our established framework, we believe it will be fairly natural to include such features into our approach in the future. We provide some thoughts about this in Section 10.

The rest of this paper is organized as follows. Section 2 presents an overview of our earlier work on formal modeling and analysis of basic statecharts using colored Petri nets, while the remaining sections address various issues central to the modeling of composite states. Section 3 introduces an intermediate state-transition notation and a transformation process for modeling the semantics of composite states. The fundamental steps for modeling entry and exit transitions are presented. In Section 4, an optimization for the translation of exit transitions is described. In Sections 5 and 6, the modeling approach is extended to include history states and completion-event semantics, respectively. Section 7 discusses some implications associated with our approach for modeling UML statecharts, and Section 8 addresses the modeling of event dispatching. Section 9 discusses related work. Finally, we provide some concluding remarks and discuss future work in Section 10.

2 Modeling and Analysis of Statecharts using Petri Nets: Overview

2.1 The Net Models

In an earlier paper [SSH01], we proposed a methodology to map UML models to Petri net models, in particular colored Petri nets (CPNs). In our methodology, statechart diagrams are converted to Object Net Models (*ONM*s), which are basically colored net models for each system object defined by a statechart diagram. The collection of ONMs defines a system-level model. The process for connecting individual object net models to create the system-level model is outside the scope of this paper, but is discussed in [SSH01]. The ONMs and system-level models are abstract net models. Using an abstract net model delays binding our transformation approach to a specific CPN notation. Thus, our transformation approach can be implemented by any standard CPN analyzer to support analysis and simulation of the

resulting CPN. Furthermore, the abstract net model allows the various issues involved in the transformation to be dealt with separately. In the first stage of the transformation, UML diagrams are converted to an abstract CPN model. This stage limits concerns to control flow and structural connections of net elements. In the second stage, the abstract net model is enriched with details related to tool-specific notations, such as defining color types, arc inscriptions, and guards for net transitions.

We now review the structure of Object Net Models (*ONM*s), as shown in Fig. 1. An *ONM* consists of two components: 1) an Event Dispatching and Processing (EDP) model and 2) an interface to other objects. The EDP model represents an abstract colored Petri net that is derived from the statechart of an object and describes the object's internal behavior, as defined by the state changes captured in the objects' statechart diagram. The interface defines two interface places – *IP* and *OP* – for exchanging tokens with other objects, and a token routing structure. Since an event of statecharts is modeled by a token in the CPN model, we use *event-tokens* to refer to the tokens derived from events. The *IP* place represents the *input place* of the object, which holds the event-tokens that will be used by the object. The *OP* place represents the *output place* of the object, which holds the event-tokens that will be routed to other objects. The *ER* place represents the *event router place*, which holds the event-tokens that are generated by the object. When the object generates an event-token, the token can have a type of either *external* or *internal*. If it is *external*, it will be routed to place *OP* via transition *T1*. Otherwise, it will be routed to place *IP* via transition *T2*. As shown in Fig. 1, place *IP* is connected to the EDP Model, indicating that *IP* holds the event-tokens that will be consumed by the object. Thus, arcs will connect the *IP* place to various transitions within the EDP model. Likewise, place *ER* is connected "from" the EDP model because *ER* holds the event-tokens that are generated by the object. So, for each transition of the EDP model, if this transition generates a new event-token, there will be an output arc targeting place *ER*. To be more precise, while it is true that the *IP*, *OP*, and *ER* places hold event-tokens, it should be understood that these event-tokens are elements of another "high-level" token that is defined as a FIFO list. Thus, when a new event-token is generated it is actually appended to the end of the list maintained by the high-level token. Likewise, when an event-token is to be routed, it is first removed from the front of the list. In the case of a sequence of events associated with a single UML transition, the sequence is preserved by appending (in order) the event-tokens to the end of the list. Further details on the event-token list are postponed until Section 8, dealing with event dispatching.

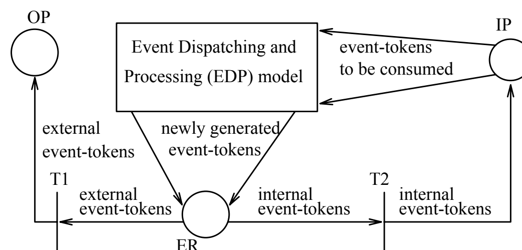


Fig. 1 The structure of Object Net Models

For the sake of illustration, we now show a very simple example of a *Power Tube* object taken from the microwave oven example [SSH01]. Fig. 2 shows the statechart for the *Power Tube* object. Initially, the *Power Tube* is in the state *Deenergized*. When an event *powerOn* occurs (generated by the *oven* object, which is not shown), the *Power Tube* object moves to state *Energized*, and a new event *lightOn* is generated. When the event *powerOff* occurs (also generated by the *oven* object), the *Power Tube* object moves back to state *Deenergized*, and a new event *lightOff* is generated. Fig. 3 shows the basic structure of the Object Net Model for the *Power Tube* object.

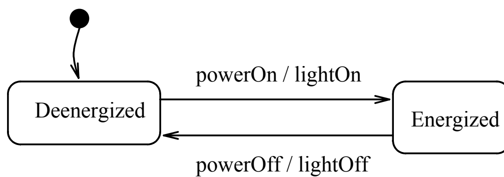


Fig. 2 Statechart for the Power Tube object

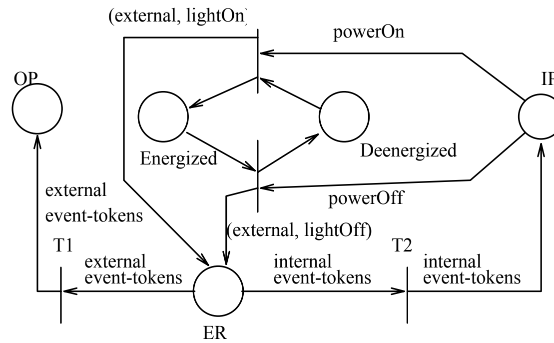


Fig. 3 *ONM* for the Power Tube object

To show that the approach advocated earlier can be realized by some existing tool, we choose the Design/CPN [DCPN] tool as an underlying engine to support analysis and simulation of UML diagrams. In earlier work [HS], we refined the framework presented in [SSH01] and presented an approach for generating a system-level target net model by enriching the abstract net model with the syntax supported by Design/CPN. In the target model, each Object Net Model is represented by a net module that defines the object behavior described initially by a UML statechart. The generated target net model can be directly imported into the Design/CPN tool for model simulation and analysis[2].

2.2 Model Analysis Based on Simulation

In general, colored Petri Net models support various model analysis techniques such as simulation and state space analysis. We use simulation to help a model designer to reason about the behavior of the system model. We have previously defined two formats of simulation results [HS]. One is a textual notation, i.e., simulation traces. The other is a graphical notation, Message Sequence Charts. The simulation reports have a text format and contain information regarding the UML transitions that occurred during the simulation. The Message Sequence Charts have an intuitive graphical appearance and are used to visualize the interaction among objects. Further details on the use and control of simulation traces are outside the scope of this paper.

---

[2] Petri nets in general, and the Design/CPN tool in particular, can support a range of analysis techniques (including model-checking), but our current work focuses on simulation.

We have developed a technique for supporting flexible user-driven automatic analysis of simulation traces. Simulations often keep log/trace files of events. Such trace files can be analyzed to check whether a run of a system model reveals faults in the system. To simulate UML statechart models, a number of commercial statechart simulators are available, such as Rhapsody [GHP02] and ObjectGEODE [OG]. We have found that these simulators provide little support for analyzing simulation traces to verify system behavioral properties. We have developed an approach for supporting flexible user-driven automatic analysis of simulation traces by providing an interface for property specification. The property specification is based on a pattern system [DAC99, DAC]. This approach has been implemented in a prototype tool called simulation query tool (SQT). We provide an interface that allows a user to construct queries regarding system properties. The queries will then be checked over simulation traces. The result for a query can be "True" or "False." In the current version of the tool, the properties that can be checked concern the occurrence and orders of events during a simulation run. As an initial investigation, three patterns of the pattern system have been implemented in the prototype tool.

Another technique concerns providing user-controlled views of system simulation through Message Sequence Charts [HS]. A complex distributed system may consist of many objects that communicate with each other through message passing. As a means to control the complexity of systems analysis, designers view systems at different levels of abstraction. To aid this process, we want a designer to be able to reason about the behavior of a subset of the objects or the occurrences of some particular events. Accordingly, an MSC can be defined to capture the behavior of a subset of the objects and/or the occurrences of some selected events. Depending on what objects and events are selected, the MSC can then provide different views for the behavior of the system. Filters are defined to tailor the views for the system behavior. Two types of filters are defined and developed: *object filters* and *event filters*. These two types of filters can be used together to control the views of system behavior. Our prototype tool provides interfaces to allow the user to choose different views of the system behavior using these filters.

2.3 Tool Support

The architecture of our UML-CPN approach is depicted in Fig. 4. The UML-CPN approach integrates four types of application software that are represented by four rectangles. Among the four applications, two are existing tools that are widely used in industry (Rational Rose) or academia (Design/CPN), and the other two are custom tools specifically developed for our approach. The role of each application plays in our approach is described as follows. The Rational Rose tool is used to design a UML statechart model. The conversion tool converts the UML model into a CPN model. The Design/CPN tool serves as the simulation and analysis engine for the generated net model. The simulation query tool supports property verification based on simulation traces.

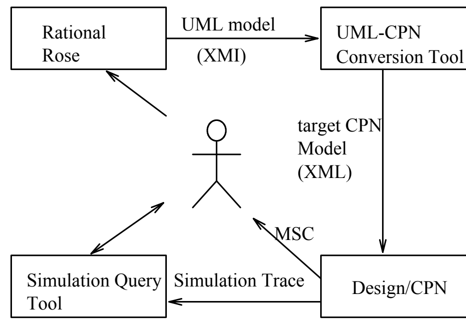


Fig. 4 A general architecture for net-based analysis of UML models

For the work discussed in this paper, the most important component of the toolset is the conversion tool, whose architecture is shown in Fig. 5. The current conversion tool does not support composite states. To handle composite states, the ONM generator must be extended to handle the many interrelated features of composite states. A key step toward this goal is to define state-transition models that make explicit the complex semantics associated with composite states, and the remainder of this paper is devoted to discussing this issue.

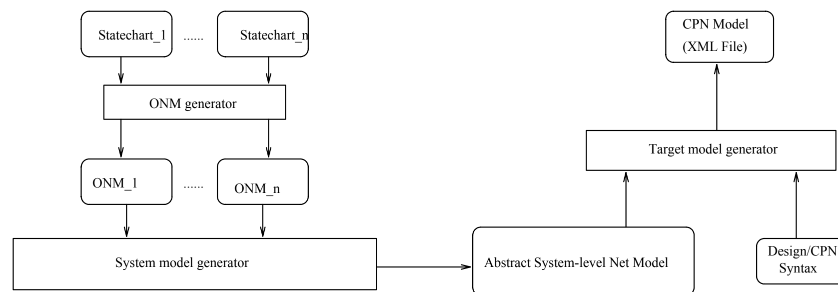


Fig. 5 The architecture of the UML-CPN conversion tool

3 A Transformation Process for Composite States

3.1 Preliminaries

In [SSH01], we presented an approach for "flattening" the hierarchical structure of very simple statecharts before converting the statecharts to net models. The technique described applied to only the most elementary form of composite states, and did not consider the key complexities associated with composite state semantics. To fully realize the potential of the UML-CPN transformation approach, it is necessary to investigate these issues and define a translation that can handle composite states. Let us consider an example adapted from [BJR99]. Fig. 6 shows a statechart that models the maintenance behavior of an ATM machine.

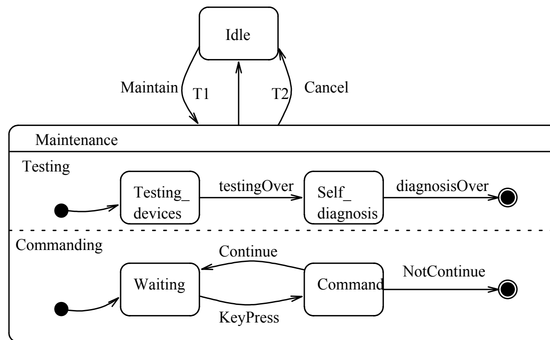


Fig. 6 A statechart containing a composite state

The statechart contains a concurrent composite state *Maintenance* that is decomposed into two concurrent substates, *Testing* and *Commanding*. Each of these concurrent substates is further decomposed into sequential substates. When the control passes from *Idle* to the *Maintenance* state, the control then forks to two concurrent flows – the object will be in the *Testing* state and the *Commanding* state. Furthermore, while in the *Testing* state, the object will be in the *Testing_devices* or the *Self_diagnosis* state; while in the *Commanding* state, the object will be in the *Waiting* or the *Command* state. When the *Cancel* event occurs, control will pass from the *Maintenance* to the *Idle* state no matter which substates the object is in. In this case, the actual source states for the transition labeled *Cancel* are not explicitly specified in the statechart. Moreover, these source states cannot be determined statically because they depend on which substates are currently active. This example gives a flavor for the complexity that we encounter when dealing with composite states.

The hierarchical structure of statecharts complicates the transformation of UML statecharts to other modeling languages because the semantics associated with composite states are often not explicitly observed in statecharts. The "implied" semantics involve the following issues: 1) transitions to and from composite states; 2) transition priority associated with the hierarchical structure of states; 3) history states; and 4) the event dispatcher mechanism. The first three issues concern the control flow of the state machine when an event is dispatched. The last issue concerns how an event can be dispatched and thus be available for driving the control flow.

Again, to retain flexibility in terms of using existing net-based tools, we define the translation to be from UML notation to an appropriate abstract net-type notation. The basic idea is to introduce new state transitions to make explicit the semantics of complex transitions associated with composite states. The new state transitions, which have a Petri net-like appearance, are derived from composite states and used as an *Intermediate State Machine* model, denoted as ISM. An ISM is a finite state machine extended with concurrency. The main idea of our approach is that *control-states* are introduced to control the flow of the state machines so that the source/target states of exit/entry transitions are explicitly represented in the ISMs. The translation to an intermediate notation allows simplifying the complex firing rules offered by

UML statecharts, through explicitly mentioning the states involved in each type of complex transition in the ISM. The intermediate model allows a decomposition of the problem by first solving hierarchical, history and composition related problems in UML statecharts before running a more straightforward translation to a state/event based formalism, here CPN. We focus on translating concurrent composite states and treat sequential composite states as special cases of concurrent composite states, i.e., a sequential composite state is a concurrent state that contains only one region.

By replacing the complex transitions with the ISM, a statechart is transformed to an *expanded state machine*. Then the expanded state machine can be converted into a CPN through direct mappings, i.e., state machine states are mapped onto Petri net places, and state machine transitions are mapped onto Petri net transitions. This resulting CPN forms part of the EDP model of the Object Net Model (*ONM*) shown in Fig. 1.

To give the reader a feel for how an ISM model is used to make explicit the semantics associated with complex transitions, we return to the example shown in Fig. 6. One requirement for correctly translating a complex transition is to ensure that the source states are deactivated and the target states are activated when the transition fires. To do so, we must directly mention the source and target states in the ISM model. For example, consider transition *T1*, which is an entry transition. When *T1* fires, the *Idle* state should be deactivated, and composite state *Maintenance* should become active. Moreover, states *Testing-devices* and *Waiting* should also become active since they are the *default states* (starting states) of the composite state. The ISM model for *T1* is shown in Fig. 7. We can see that the source state for *T1* is *Idle* and the target states are the composite state *Maintenance* itself and its two default states.

In our notation for an ISM model, a transition is labeled with a transition name and event name, separated by a colon. Both name and event are optional. For example, Fig. 7 shows a transition called *init* that has a triggering event *Maintain*.

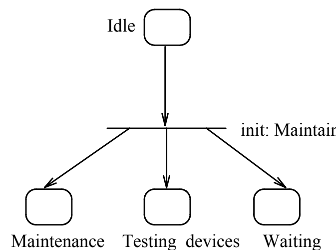


Fig. 7 Model for entry transition *T1* in Fig. 6

3.2 Composite States: Definitions and Translation Strategy

For the sake of establishing common terminology, we start with some definitions for the different types of UML states and transitions, and some definitions that relate closely to our scheme for composite state translation. For illustration, consider the statechart shown in Fig. 8. We use this somewhat contrived

example as a "running" example since it illustrates in one example each of the key features to be considered throughout the paper.

Fig. 8 shows a statechart containing a concurrent composite state with two parallel state machines *S1* and *S2*. An *initial state* is represented as a filled black circle. There is a transition that originates from an initial state and targets some state of the state machine or substate. This target state is called a *default state*, which is the default starting state for the state machine or substate. For example, state *I1* is the default state of the state machine; states *A* and *C* are the default states of composite state *X*. A *final state* indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented by a filled black circle surrounded by an unfilled circle, as seen in Fig. 8.

A *simple transition* is a transition that has only one source state and one target state and these states are simple states. A *complex transition* is a transition that enters to, or exits from, a composite state. Transitions that are nested within a composite state are called *nested transitions*. In Fig. 8, transitions *T5 T6*, *T9*, and *T10* are simple transitions (also nested transitions), while transitions *T1*, *T2*, *T3*, *T4*, *T7*, and *T8* are complex transitions. A *boundary transition* is a complex transition that leads to and/or emanates from the boundary of a composite state. For example, *T1*, *T2*, and *T7* are boundary transitions. A *cross-boundary* transition is a complex transition that targets, or originates from, the nested state(s) of a composite state. For instance, *T3*, *T4*, and *T8* are cross-boundary transitions.

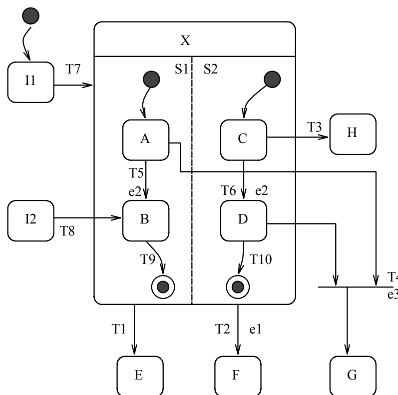


Fig. 8 A statechart containing a concurrent composite state

Complex transitions can also be categorized as *entry transitions* or *exit transitions*. An *entry transition* is a transition that leads directly to a composite state. When an entry transition fires, the composite state is activated. For example, *T7* and *T8* in Fig. 8 are entry transitions. If *T7* or *T8* fires, state *X* is entered. Graphically, there are two types of entry transitions, those that reach a composite state boundary, such as *T7*, and those that reach the nested state(s) of a composite state, such as *T8*. An *exit transition* is a transition that emanates from a composite state. As a result of firing an exit transition, a composite state is exited. In Fig. 8, *T1*, *T2*, *T3*, and *T4* are exit transitions. Graphically, there are two types of exit transitions, those that originate from a composite state boundary, such as *T1* and *T2*, and

those that originate from the nested state(s) of a composite state, such as *T3* and *T4*. For example, *T3* originates from a nested state *C*, however, when *T3* fires, the entire composite state *X* is exited. This graphical notation denotes that *T3* is enabled no matter what nested state of substate *S1* is active, given that state *C* is active and event *e4* occurs.

As indicated before, translation of entry and exit transitions of composite states are the main focus of this paper. A key concept associated with composite states is: *configuration*. When a composite state is active, a set of nested states of the composite state are also active. We call the set of active states a *configuration* of the composite state. Besides the composite state itself, a configuration of a *sequential composite state* also contains one nested state, while a configuration of a *concurrent composite state* contains more than one nested state, one from each orthogonal region of the composite state. For illustration, consider Fig. 8. The set {*X*, *B*, *C*} defines one configuration of state *X*. In UML, the regions that compose a concurrent composite state are also considered as composite states; thus regions are considered states. However, for the sake of simplicity, we do not include regions directly in state configurations.

When considering an exit transition, the source state is a composite state, which can have different configurations depending upon which nested states can become active. For example, in Fig. 8, composite state *X* can have the following nine configurations: {*X*, *A, C*}, {*X, A, D*}, {*X, A, S2_Fin*}, {*X, B, C*}, {*X, B, D*}, {*X, B, S2_Fin*}, {*X, S1_Fin, C*}, {*X, S1_Fin, D*}, and {*X, S1_Fin, S2_Fin*}, where *S1_Fin* and *S2_Fin* denote the final states of the regions *S1* and *S2*, respectively. For brevity, we use {*X, A*|*B*|*S1_Fin, C*|*D*|*S2_Fin*} to represent the nine configurations. We use "*A*|*B*|*S1_Fin*" to denote state *A* or state *B* or state *S1_Fin*. Some or all of the configurations can identify source states of the exit transitions. For transition *T3*, {*X*, *A*|*B*|*S1_Fin, C*} are configurations that define source states. We call such configurations *source configurations* of the exit transition. Similarly, the target states of an entry transition include a composite state and some nested states of the composite state. A *target configuration* of an entry transition is a configuration that can potentially define the target states of the entry transition. For example, in Fig. 8, the target configuration for transition *T7* is {*X, A, C*}. However, in contrast to an exit transition, an entry transition has only one target configuration.

The goal for translating an entry and an exit transition is to generate an intermediate model that consists of three key components: 1) source and target configurations of the transition, 2) a control flow model that starts from the source configuration and ends at the target configuration, and 3) a trigger for driving the control flow. In this way, the semantics of the entry and exit transitions become explicitly presented in the intermediate models.

In order to define source and target configurations, we follow the UML standard [OMG03]. Entry and exit transitions have many variations in UML – some are boundary transitions, while others are cross-boundary transitions; and some are *completion transitions*, while others are *triggered transitions* (having

an explicit triggering event). Since different variations usually imply different configurations, we define our transformation approach for each type of entry/exit transition. To define the control flow for an entry or exit transition, we distinguish whether or not the transition is a completion transition. To handle a completion transition, we must deal with the completion-event semantics of the completion transition, i.e., a completion event must be generated to trigger the completion transition. For convenience and separation of concerns, in the following several sections we will treat completion transitions as if they have an implicit trigger, which we will call a *completion trigger*. According to UML semantics, this implicit trigger is actually a completion event that must be explicitly generated. Generation of completion events is discussed in Section 6.

We first illustrate the transformation via examples. Then we provide basic semantic rules to summarize the semantics that we capture in our transformation process and to provide a basis for automating the transformation.

3.3 Modeling Entry Transition Semantics

In this section, we show how to translate entry transitions. As the UML standard [OMG03] describes, when an entry transition of a concurrent composite state fires, the composite state is entered; each one of its regions is also entered, either by default or explicitly. If a region is entered by default, its *default state* is entered. If a region is entered explicitly, the nested state specified explicitly by the entry transition is entered.

We first consider cases where the source configuration of an entry transition is a simple state[3]. To model a UML entry transition, a *fork transition* is defined to make explicit the control flow for the entry transition. A fork transition has one source state and multiple target states. We call this special fork transition an *init transition*. An *init* transition uses an explicit trigger to model the triggering-condition for an entry transition. The source state of the *init* transition is the same as that of the UML entry transition. The target states of the fork transition include a simple state representing the composite state and some *control-states*, one for each concurrent region. Control-states, which have no direct counterpart in UML statecharts, are introduced as the technical vehicle for driving the control flow for entering concurrent regions. Note that a control-state is a simple state. Furthermore, for each region, a transition is defined for entering some nested state within the region. The source state of the transition is the control-state associated with the region. Thus, an entry transition is translated to a set of transitions that constitutes a tree structure, with the *init* transition serving as the root.

For illustration, we translate the entry transitions in Fig. 8, transitions *T7* and *T8*. *T7* is a boundary entry transition while *T8* targets a nested state *B*. *T7* and *T8* are represented as follows:

[3] Modeling of entry transitions whose source states are also composite states is discussed in Section 3.5.

$$T7 = I1 \longrightarrow X, \qquad T8 = I2 \longrightarrow X(B),$$

where we use *X (B)* to denote the nested substate (state *B*) of composite state *X*.

Since *T7* is a boundary entry transition, when *T7* fires, state *X* is entered; and regions *S1* and *S2* are also entered, by default. Fig. 9 depicts the model for transition *T7*. As mentioned, the model has a tree structure where an *init* transition and two *control-states, D_En_S1 and D_En_S2,* are introduced. Since *T7* is a completion transition, event *cI1* represents the (implicit) completion trigger for transition *T7*. In our intermediate model, a control-state is represented as an oval. State *D_En_S1* (Default Entry State *S1*) represents the fact that region *S1* will be entered by default. This is modeled by transition *t1*. Similarly, state *D_En_S2* represents the fact that region *S2* will be entered by default. This is modeled by transition *t2*. As a result, states *A* and *C* are entered.

Fig. 10 depicts the model for transition *T8*, which is a cross-boundary entry transition. Compared to translation of *T7*, the only difference is that region *S1* is entered explicitly instead of by default. As shown in Fig. 10, the control-state *E_En_S1_B* (Explicit Entry State S1 due to State B) represents the fact that region *S1* will be entered explicitly due to state *B*. Since *T8* is a completion transition, event *cI2* represents the completion trigger for transition *T8*. It is easy to observe that the models of Fig. 9 and Fig. 10 can be simplified by removal of the control-states, allowing a direct activation of the target states. For example, Fig. 9 can be simplified/reduced so that the *init* transition has three output states: *X*, *A*, and *C*. A generalized reduction rule for such cases is presented at the end of this section.

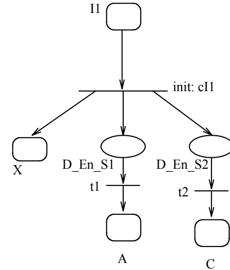


Fig. 9 Translation of entry transition *T7* in Fig. 8

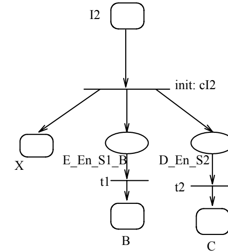


Fig. 10 Translation of entry transition *T8* in Fig. 8

As a result, when transition *init* in Fig. 9 fires the control-states will force the model to reach a state configuration in which states *X*, *A*, and *C* are active. This defines the target configuration {*X*, *A*, *C*} for transition *T7*. When transition *init* in Fig. 10 fires the following states are entered: *X*, *B*, and *C*, which define the target configuration {*X*, *B*, *C*} for transition *T8*.

Semantic Rule 3.1: Default Entry. Given a concurrent composite state *X* with a set of regions *R*: $\forall\ r_i \in R$, r_i has a default state I_i and if r_i is entered by default, I_i is entered. If I_i is a concurrent composite state with a set of regions R_I, then $\forall\ r_j \in R_I$, r_j is entered by default.

Semantic Rule 3.2: 1-level Explicit Entry. Given a concurrent composite state *X* with a set of regions *R*: $\forall r_i \in R$, if r_i is entered explicitly due to some state *A* that is directly (1-level) nested within r_i, then state *A* is entered. If *A* is a concurrent composite state with a set of regions R_A, then $\forall r_j \in R_A$, r_j is entered by default.

Semantic Rule 3.2': Multilevel Explicit Entry (p-level Explicit Entry, p>1). Given a concurrent composite state *X* with a set of regions *R*: $\forall r_i \in R$, if r_i is entered explicitly due to some state *A* that is *p*-level nested within region r_i, then $\exists$ concurrent composite state *S* with a set of regions R_S, such that *S* is directly nested within r_i, and $\exists r_A \in R_S$ such that r_A contains state *A*, and 1) state *S* is entered; 2) r_A is entered via (*p*-1)-level explicit entry; and 3) $\forall r_j \in R_S$, such that $r_j \neq r_A$, r_j is entered by default.

Semantic Rule 3.3: Boundary Entry Transition. Given a concurrent composite state *X* with a set of regions *R* and an entry transition *t* such that *t* is a boundary entry transition: If *t* fires, state *X* is entered and $\forall r_i \in R$, r_i is entered by default (Rule 3.1).

Semantic Rule 3.4: Cross-Boundary Entry Transition[4]. Given a concurrent composite state *X* with a set of regions *R* and an entry transition *t* that targets nested state *A* belonging to region $r_i \in R$: If *t* fires, then 1) state *X* is entered; 2) if state *A* is directly nested with region r_i, region r_i is entered via 1-level explicit entry (Rule 3.2); otherwise, region r_i is entered via multilevel explicit entry (Rule 3.2'); and 3) $\forall r_j \in R$ such that $r_j \neq r_i$, r_j is entered by default (Rule 3.1).

We briefly discuss a more complex case that involves *multilevel composite states*. A composite state that contains yet other composite states is called a multilevel composite state. To translate entry transitions of multilevel composite states, we need to apply Rules 3.1 – 3.2 multiple times until the innermost states are entered. In other words, if the nested state being entered, call it *S*, is a concurrent composite state, *S* is entered and also each orthogonal region of *S* is entered by default. As a contrived example, consider what would happen if state *A* in Fig. 8 were a concurrent composite state, such as that shown in Fig. 11, rather than a simple state. Now, when we translate entry transition *T7* in Fig. 8 (Rule 3.3), we would obtain the structure shown in Fig. 12. Note that the transition *t11* represents the entry to the composite state *A*, and the model shows how to reach the target configuration of *T7*, {*X, A, A1, A3, C*}.

[4] For simplicity, we deal with entry transitions that point to one nested state. However, the approach can be conveniently extended to cases that an entry transition is a fork transition and thus targets multiple nested states.

As mentioned earlier, sometimes ISM models can be simplified by removing redundant transitions and associated control-states if certain conditions are satisfied. For illustration, consider the control-states in Fig. 12. Only transition *t11* targets control-state *D_En_R1*; only transition *t12* originates from *D_En_R1*; and state *D_En_R1* is the only source state of transition *t12*. Thus, control-state *D_En_R1* can be eliminated, and transitions *t11* and *t12* can be merged. Likewise, control-states *D_En_R2*, *D_En_S1,* and *D_En_S2* can also be eliminated. The simplified resulting model is shown in Fig. 13.

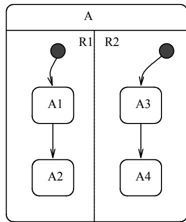


Fig. 11 Concurrent composite state *A*

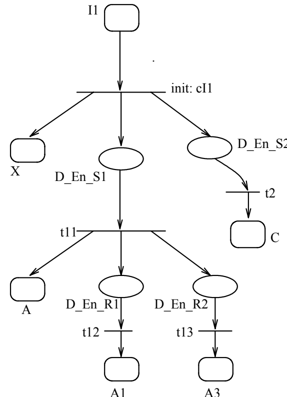


Fig. 12 Translation for entry transition *T7* with state *A* being a concurrent composite state

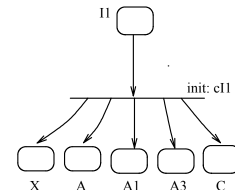


Fig. 13 Reduction applied to Fig. 12

Translation Rule 3.5: Control-State Reduction Rule. Given a control-state *S* such that 1) There is exactly one transition *t1* targeting *S*; 2) There is exactly one transition *t2* originating from *S* and *t2* is not a triggered transition; and 3) *S* is the only source state of *t2*: *S* can be eliminated, and transitions *t1* and *t2* can be merged into one transition.

3.4 Modeling Exit Transition Semantics

In this section, we show how to translate exit transitions for each of the basic cases: boundary exit transitions, and cross-boundary exit transitions, both with and without triggers. As the UML standard [OMG03] describes, when exiting from a concurrent state, each of its regions is exited, i.e., the currently active nested state of each region is exited. Thus, in general, each exit transition corresponds to a set of state transitions in our model. These state transitions differ in terms of their source states, which depend on source configurations.

To make the semantics of exiting a composite state explicit, we explicitly define the source configurations. There are four different cases: 1) A boundary exit transition with an explicit trigger; 2) A boundary exit transition that does not have an explicit trigger; and 3) A cross-boundary exit transition with an explicit trigger; and 4) A cross-boundary exit transition without an explicit trigger.

First, we consider a boundary exit transition with an explicit trigger. Such transition is enabled if the composite state is active and the triggering event is generated, no matter which substates of the composite state are active. So the source configurations of the exit transition involve the combinations of the nested states of the concurrent regions. The model for this exit transition actually represents a set of state transitions. Each of the state transitions has the same target state and triggering event, while the source states for each transition are defined by one of the aforementioned source configurations.

A boundary exit transition that does not have an explicit trigger is enabled when the control flow for each orthogonal region of the composite state reaches the final state. Thus, the source configuration of this type of exit transition consists of the composite state itself and the final state of each orthogonal region of the composite state.

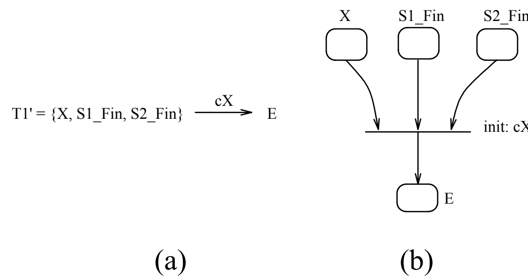
A cross-boundary exit transition with or without an explicit trigger can be treated similarly as the case of a boundary exit transition with a trigger. The difference is that for a cross-boundary exit transition the source configurations involve the combinations of the nested states of a *subset* of the concurrent regions. In addition, a cross-boundary exit transition that does not have an explicit trigger is enabled when the control flow is ready to leave the nested state that is the direct source of the exit transition.

For illustration, we translate the exit transitions, *T1*, *T2*, *T3* and *T4*, in Fig. 8. These transitions are the typical cases of interest. These four transitions are represented as follows:

$$T1 = X \longrightarrow E \qquad T2 = X \xrightarrow{e1} F$$

$$T3 = X(C) \longrightarrow H \qquad T4 = \{X(A), X(D)\} \xrightarrow{e3} G$$

Consider transition *T1*, which is a boundary exit transition that does not have an explicit trigger (a *completion transition*). As mentioned earlier, we assumed that a completion transition has a completion trigger; here, we denote this trigger as *cX*. The source configuration for *T1* is {*X*, *S1_Fin*, *S2_Fin*}, where *S1_Fin* and *S2_Fin* denote the final states of regions *S1* and *S2*, respectively. So, *T1* is translated to *T1'* shown in Fig. 15 (a). The graphical notation for model *T1'* is shown in Fig. 15 (b). As in the case for entry transitions, a net transition, called *init*, is defined to recognize when an exit transition is to be enabled. For example, in Fig. 15(b), the *init* transition is enabled when 1) composite state *X* itself and final states *S1_Fin* and *S2_Fin* are active; and 2) completion trigger *cX* occurs.



(a) (b)

Fig. 15 Translation for exit transition *T1* in Fig.8

Now consider transition *T2*, which is a boundary exit transition with trigger *e1*. The source configurations of *T2* involve the combinations of the nested states of regions *S1* and *S2*. This defines nine source configurations, denoted as {*X, A|B|S1_Fin, C|D|S2_Fin*}. Thus, the model for transition *T2* consists of nine transitions. So, *T2* is translated to *T2'*, as shown in Fig. 16 (a). For illustration, we show the graphical notation for four of the nine state transitions of *T2'* in Fig. 16 (b).

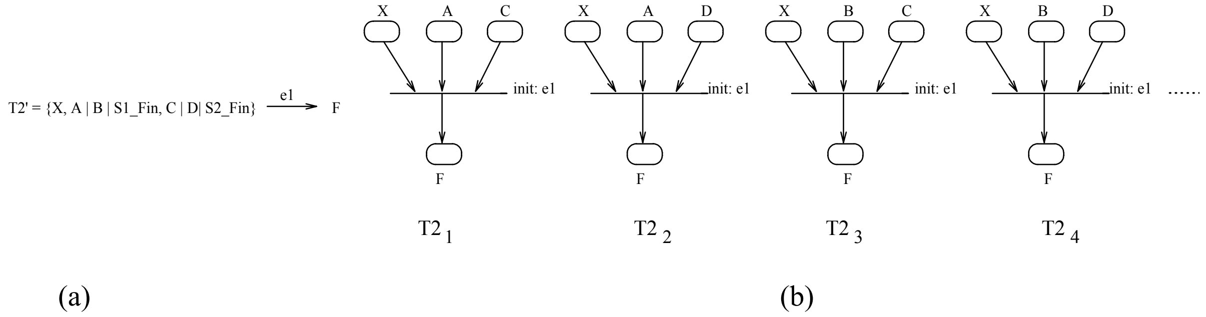


Fig. 16 Transitions derived from exit transition *T2* in Fig.8

Transition *T3* is a cross-boundary exit transition that does not have an explicit trigger. *T3* is enabled if the control flow is ready to leave state *C* and any nested state of region *S1* is active. Since *T3* is also a completion transition, a completion trigger *cC* is defined. The source configuration for *T3* is {*X*, *A*|*B*|*S1_Fin*, *C*}. The transition *T4* is also a cross-boundary exit transition with trigger *e3*. *T4* is enabled if states *A* and *D* are active and event *e3* is generated. The source configuration of *T4* is {*X*, *A*, *D*}. Thus, transitions *T3* and *T4* are translated into *T3'* and *T4'*, respectively. *T3'* and *T4'* are shown as follows. The graphical notations for *T3'* and *T4'* will be similar to that of *T2'*.

The transformation approach for exit transitions is defined in the following semantic rules. The translation of *T1* is an example that relates to Rule 3.6. Similarly, Rule 3.7 is for transitions like *T2*, and Rule 3.8 is for transitions like *T3*.

Semantic Rule 3.6: Boundary Exit Transition without Trigger. Given a concurrent composite state *X* with a set of regions *R*, where $|R| = n$, and an exit transition *t* such that *t* is a boundary exit transition and *t* is a completion transition: $\forall\ r_i \in R$, $\exists$ state f_i, such that f_i is the final state of r_i, and the source configurations of transition *t* are a set of configurations $SC = \{\{X, f_1, f_2, \ldots, f_n\}\}$.

Semantic Rule 3.7: Boundary Exit Transition with Trigger. Given a concurrent composite state *X* with a set of regions *R*, where $|R| = n$, and an exit transition *t* such that *t* is a boundary exit transition and *t* has a triggering event: $\forall\ r_i \in R$, $\exists\ S_i$, such that S_i is the set of the nested states of region r_i, and the source

configurations of transition t are a set of configurations $SC = \{\{X, e_{i1}, e_{i2}, \ldots, e_{in}\} \mid e_{ij}$ is the jth element of $p_i \in S_1 \times S_2 \times \ldots \times S_n, i = 1..(|S_1| \cdot |S_2| \cdot \ldots \cdot |S_n|), j = 1..n \}$.

Semantic Rule 3.8: Cross-Boundary Exit Transition[5]. Given a concurrent composite state X with a set of regions R, where $|R| = n$, and an exit transition t that originates from nested state A belong to region r_i: $\forall\ r_j \in R, \exists\ S_j$, such that S_j is the set of the nested states of region r_j, and the source configurations of transition t are a set of configurations $SC = \{\{X, A, e_{i1}, e_{i2}, \ldots, e_{i(n-1)}\} \mid e_{ik}$ is the kth element of $p_i \in S_1 \times S_2 \times \ldots \times S_{i-1} \times S_{i+1} \times \ldots \times S_n, i = 1..(|S_1| \cdot |S_2| \cdot \ldots \cdot |S_{i-1}| \cdot |S_{i+1}| \cdot \ldots \cdot |S_n|), k = 1..(n-1) \}$.

Note that the number of state transitions resulting from translating an exit transition can be very large due to the various combinations of nested states that can be used as source states of the resulting state transitions. To address this issue, we will present an optimization technique for translation of exit transitions in Section 4. Once the optimization technique is presented, we will consider exit transitions whose source states are multi-level composite states.

3.5 Modeling a Transition with Dual-Role Semantics

As an extended case, we consider transitions whose source and target states are both composite states. To model such a transition we divide the original transition into two transitions, call them *T1* and *T2*, by adding a control-state. Transition *T1* is an exit transition that originates from the source state of the original transition and targets the control-state; and transition *T2* is an entry transition that originates from the control-state and targets the target state of the original transition. Note that transition *T1* has the same enabling condition as the original transition, while transition *T2* is a *triggerless transition*. Because the control-state is a simple state, transitions *T1* and *T2* can be modeled using the approaches previously described.

It is noteworthy that a control-state is not a state in the original specification and thus the control flow should not "stop" in this intermediate state. Our model guarantees this by ensuring that the firings of transitions *T1* and *T2* are included in one run-to-completion step. (Further details on the run-to-completion step are discussed in Section 8.) Therefore, once the enabling condition for the original transition is satisfied and triggers transition *T1* (since transition *T1* has the same enabling condition as the original transition), *both T1* and *T2* will fire. Thus, it is not possible for the net model to "stop" in the intermediate state.

[5] For simplicity, we present Rule 3.8 for dealing with exit transitions that originate from one nested state. However, the approach can be extended to cases where an exit transition is a join transition and thus originates from more than one nested state.

4 A Translation Optimization for Exit Transitions

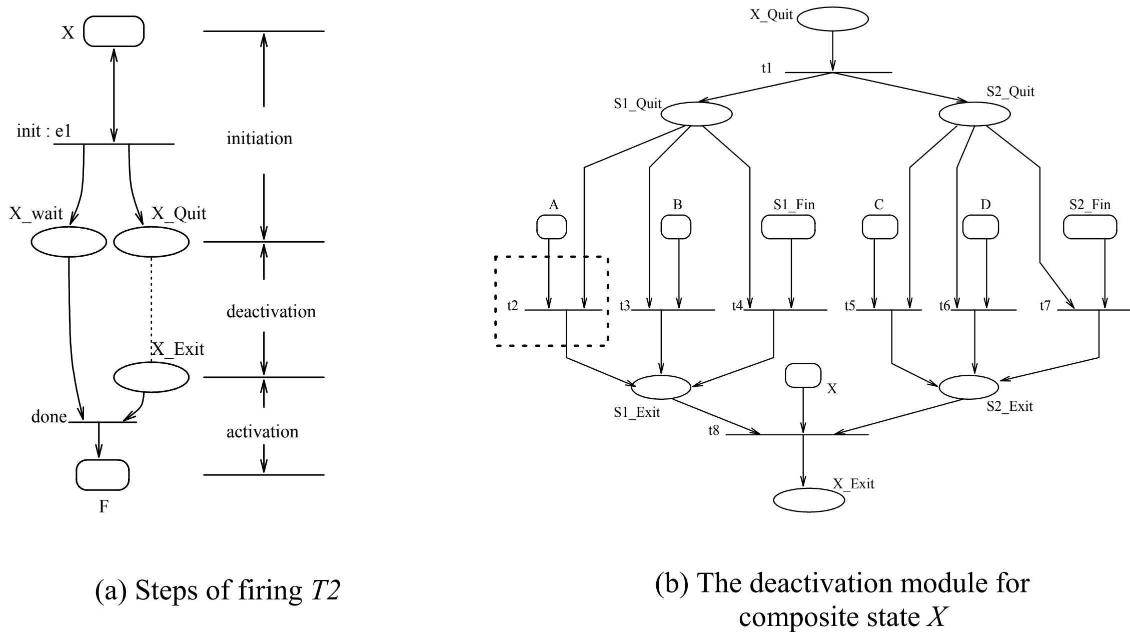
In the case of translating exit transitions, if a concurrent composite state has more than two regions or each region has a large number of nested states, the number of state transitions obtained from translating the exit transition of this composite state can be very large. For illustration, assume a concurrent composite state has *m* orthogonal substates and each substate has *n* nested states. In general, a triggered transition that originates from the boundary of the concurrent composite state would be translated into n^m transitions. More specifically, consider state *X* in Fig. 8. This state has 2 orthogonal regions *S1* and *S2*. Each region has 3 nested states, including the final states. The resulting transition list for exit transition *T2* contains 9 state transitions described previously in Section 3.4. In this section, for the sake of optimizing the translation approach, we present a method for decreasing the number of the state transitions obtained from the translation of the exit transitions associated with composite states.

The key idea of the optimization approach is that the initiation of firing an exit transition is separate from the deactivation of its source states. More specifically, the firing of an exit transition is divided into three steps. The first step, call it *initiation*, is to recognize when an exit transition is to be enabled. For example, in Fig. 8, transition *T2* is enabled when state *X* is active and event *e1* has occurred. Likewise, exit transition *T4* is enabled when states *A* and *D* are active and event *e3* has occurred. The second step, call it *deactivation*, is to deactivate the source states. For example, in Fig. 8, when transition *T2* fires, the following states are deactivated: state *X*, and one of the nested states in each concurrent region. A *deactivation module* is defined to deactivate the source states for exit transitions associated with a composite state. The third step, call it *activation*, is to activate the target state. For example, in Fig. 8, when transition *T2* fires, state *F* is activated.

In the initiation step, if the *direct* source state(s) is active and the triggering event is present, a *control-state* "*Quit*" becomes active, which drives the deactivation of the source states. The "*Quit*" state activates other "*Quit*" states, as many as one for each orthogonal region. The activeness of these "*Quit*" states leads to the deactivation of the currently active substates. After the substates are deactivated, the composite state is deactivated. After the source states are deactivated, the target state is activated. Note that the direct source state(s) of a transition means the source state(s) that is explicitly specified in the statechart diagram, unless the transition is a completion transition. In this case, the direct source states are the final states of the regions. For example, in Fig. 8, the direct source state of *T3* is state *C*, while the direct source states of *T1* are *S1_Fin* and *S2_Fin*. This approach for modeling exit transitions is motivated by the technique outlined in [DFH03], where an "*abort*" token is injected into a CPN model to abort a state within a composite state.

Let us illustrate via the same example as above, transition *T2* in Fig. 8. Assume that state *X* is active and event *e1* occurs. Transition *T2* is enabled and fires. The three steps of firing transition *T2* are depicted

in Fig. 17. More specifically, Fig. 17 (a) illustrates Step 1 (initiation) and Step 3 (activation), and the starting and ending points of Step 2. The details of Step 2 (deactivation) are shown in Fig. 17 (b).



(a) Steps of firing *T2*

(b) The deactivation module for composite state *X*

Fig. 17 Translation of exit transition *T2* in Fig.8

In Step 1, if state *X* is active and event *e1 occurs*, transition *init* is enabled. Note that the enabling condition of transition *init* in Fig. 17 (a) is the same as that of *T2* in Fig. 8. When *init* fires, states *X_Quit* and *X_wait* become active. The activeness of state *X_Quit* leads to the beginning of the second step, the step of deactivation. Note that, implicitly, some substates of regions *S1* and *S2* are active when the enabling condition of *T2* is evaluated. However, Fig. 17 (a) does not show which substates are active when the enabling condition is evaluated. This means that the specific substates that are active are not significant in terms of enabling transition *T2*. This fact simplifies the step of initiation. Moreover, it allows the UML statechart to serve explicitly as the direct basis for translation in Step 1. The state *X_wait* is defined as a "place holder." Because the deactivation module is designed to be shared by all exit transitions associated with the composite state, a place holder is defined to remember which specific exit transition should complete when the deactivation is done. Note that the arc connecting state *X* to transition *init* is a bi-directional arc. The arrow pointing to state *X* represents that state *X* is still active after Step 1. Only in Step 2 is it deactivated. There are two reasons for this design. First, the deactivation module can have a generic form and be reused by all the exit transitions of the composite state. The second reason is to separate the issue of evaluating the enabling condition from that of deactivating the source states.

Step 2, simplified as a dashed line in Fig. 17 (a), is illustrated in Fig. 17 (b). States with the same name should be treated as the same state, such as *X_Quit* in Fig. 17 (a) and (b), and *X_Exit* in Fig. 17 (a)

and (b). When *X_Quit* is active, transition *t1* is enabled. When *t1* fires, control-states *S1_Quit* and *S2_Quit* become active. Transitions *t2 – t7* model the deactivation of the substates. When *S1_Quit* becomes active, *one and only one* of the three transitions originating from *S1_Quit* is enabled; when one of the transitions (*t2*, *t3*, or *t4*) fires, the currently active substate of region *S1* is deactivated. Likewise, the currently active substate of region *S2* is deactivated via the firing of one of the transitions originating from *S2_Quit*. When transition *t8* fires, state *X* is deactivated, resulting in state *X_Exit* becoming activated. This is the end of Step 2 and the beginning of Step 3. Now, the control flow goes back to the model shown in Fig. 17 (a). Since states *X_Exit* and *X_wait* are active, transition *done* is enabled. When *done* fires, state *F* is activated, which means the target state of transition *T2* has been entered. This concludes the three steps of exit transition *T2*.

Fig. 18 shows the translation of the other three exit transitions in Fig. 8. Since all the exit transitions share the same deactivation module, the details of this module are not shown in Fig. 18. Interested readers can refer to Fig. 17 (b) for the details. The module presented in Fig. 17 (b) does not require using each combination of nested states as source states for the resulting transitions. Thus, the number of resulting transitions is decreased. Actually, the number of transitions in the deactivation module is linear to the number of nested states.

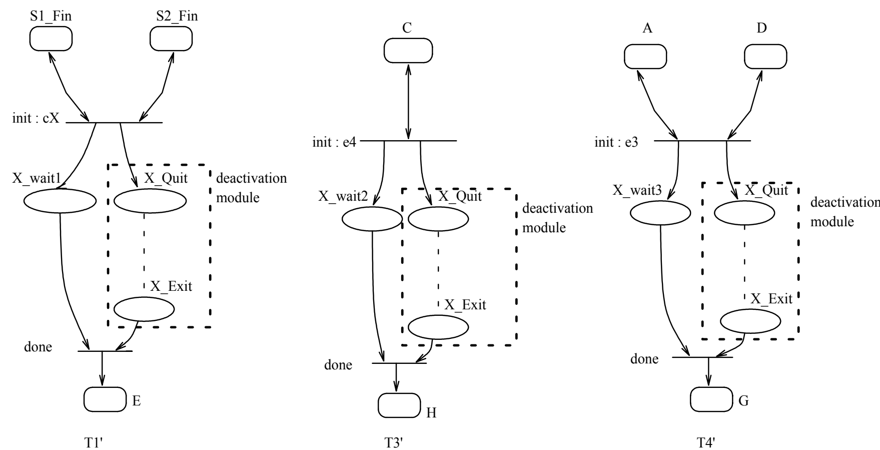


Fig. 18 Translation of exit transitions *T1*, *T3*, and *T4* in Fig.8

We presented an approach to decrease the number of transitions obtained from translating an exit transition. The reason is that it is well understood that expanding a hierarchical model usually causes a state explosion problem. This occurs when the original model contains concurrent constructs and/or has multiple levels of nested structure. The optimization approach provides a generic form for translation of exit transitions. Besides unifying the form of the resulting model, the generic form supports the translation of exit transitions of composite states that have a multilevel hierarchical structure.

In case of multilevel composite states, a nested state itself can be a composite state. Thus, in order to deactivate a multilevel composite state, multiple deactivation modules are used, one for each composite

state associated with the multilevel composite state. As a contrived example, let's consider what would happen if state *A* in Fig. 8 were a concurrent composite state. In this case, state *X* becomes a multilevel composite state. Assume that state *A* is active when transition *T2* fires. In Fig. 17 (b), where state *A* is a simple state, transition *t2* deactivates state *A*. But, since state *A* is now a composite state, a deactivation module is needed to deactivate state *A*. For simplicity, we do not present this deactivation module, which has the same form as shown in Fig. 17 (b). Fig. 19 shows that control-state *S1_Quit* drives the deactivation of composite state *A*. Accordingly, the modification needed for the original model shown in Fig. 17 (b) is to replace transition *t2* with the sub net in the dashed-rectangle in Fig. 19.

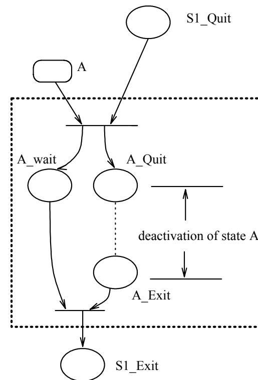


Fig. 19 Deactivation of composite state *A*

As a conclusion to the translation of exit transitions, we summarize the above optimization approach in the following translation rule.

Translation Rule 4.1: Basic Structure for Exit Transitions. Given a concurrent composite state *X* with a set of regions *R* and an exit transition *t*: The model for transition *t* consists of an *initial* transition for recognizing that transition *t* is enabled, a *deactivation* module for deactivating the source states, and an *activation* transition for activating the target state.

5 History States and Revisiting Entry and Exit Transitions

In UML, a *history state* allows a composite state that contains sequential substates to remember the most recently active substate prior to exiting from the composite state [BJR99]. Two kinds of history states are defined in UML, *shallowHistory* and *deepHistory*. A shallowHistory, represented as a small circle containing the symbol H, remembers only the history of the immediate nested state machine. A deepHistory, represented as a small circle containing the symbol H*, remembers down to the innermost nested state at any depth. If the composite state has only one level of nesting, shallow and deep history states are semantically equivalent. Note that we follow the statement given in [BJR99]: A nested

concurrent composite state does not have a history state; however, the orthogonal regions that compose a concurrent composite state may have one.

A history state relates directly to entry transitions and exit transitions of a composite state. The modeling of the history state in a region requires some form of memory to know which substate of that region was active immediately before some exit transition fires. To model history states we introduce the idea of a *shadow state* for each nested state of the region. If some nested state is active when an exit transition fires, the associated shadow state is activated. Consequently, the translation of exit transitions needs to be modified to remember the history via shadow states, which will then be used to guide the history entries into the composite state. Suppose that an entry transition targets a history state of an orthogonal region. When the entry transition fires, the most recently active substate of the region is entered, unless the most recently active substate is the final state or the region is being entered for the first time. In the latter two cases, the *default history state* is entered. The *default history state* is the substate that is the target of the transition originating from the history state.

Here, we deal with composite states that have only one level of nesting, so only shallow history states are involved. However, the approach can be extended to handle deep history states in the context of modeling multilevel composite states. As mentioned in Section 3.3, when an entry transition of a concurrent composite state fires, there are two ways of entering the orthogonal regions of the composite state: *default entry* and *explicit entry*. Now, history states introduce a new way of entering orthogonal regions: *history entry*. To model that a region is entered via history entry, the nested state whose shadow state is currently active becomes activated.

We illustrate our approach via an example. Fig. 20 is the same as Fig. 8 except that region *S1* has a history state and a new transition *T11*, which originates from state *I3* and targets the history state. State *A* is the default history state. Accordingly, the deactivation module (shown in Fig. 17 (b)) is modified to remember the history of region *S1*; See Fig. 21. Consider transition *T4* in Fig. 20. Assume that *T4* is enabled, which means that states *A* and *D* are active. When *T4* fires the three steps of modeling exit transitions are carried out. Fig. 21 depicts the deactivation module of composite state *X*. When state *S1_Quit* becomes active, transition *t2* is enabled. When transition *t2* fires state *A* is deactivated. At the same time, the shadow state *A'* is activated. Thus, the shadow state remembers the history of region *S1*.

A final state does not have a shadow state of its own. If the currently active state of a region is the final state, the shadow state of the default history state (in this case, state *A* is the default history state) is activated. Assume that, in Fig. 21, the final state of region *S1*, *S_Fin*, is active when the composite state is exited. Thus, transition *t4* is enabled when state *S1_Quit* becomes active. When transition *t4* fires, shadow state *A'* is activated.

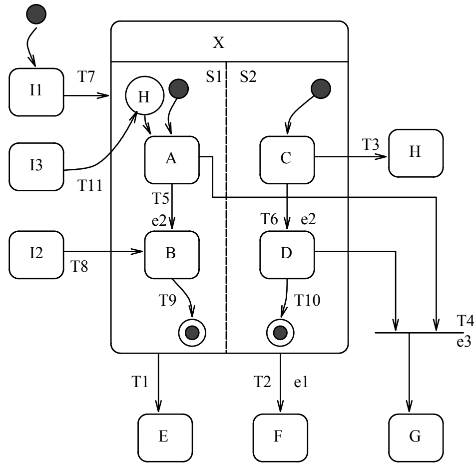


Fig. 20 A concurrent composite state with history state

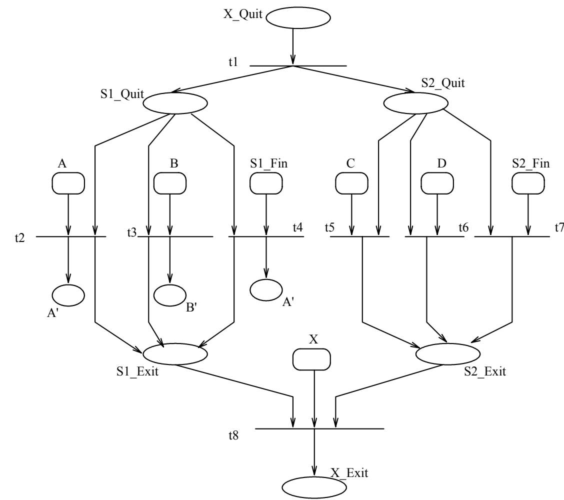


Fig. 21 Deactivation module of composite state *X*

We can now proceed to model transition *T11* in Fig. 20. This transition is an entry transition that targets a history state. Since transition *T11* targets the history state of *S1*, history entry is used for entering *S1* and default entry is used for entering *S2*. The model for transition *T11* is shown in Fig. 22. Note that the model is a 2-level model. Event *cI3* represents the (implicit) completion trigger for *T11*, since *T11* is a completion transition. When transition *init* fires, composite state *X* is activated; some substate of region *S1* is entered via history entry; and the default state of region *S2*, i.e. state *C*, is entered via default entry (Note that the Control-state Reduction Rule (Rule 3.5) has been used to remove redundant control-states). When control-state *H_En_S1* (<u>H</u>istory <u>En</u>try State <u>S1</u>) is active, either transition *H_En_A* or *H_En_B* is enabled depending on what shadow state, *A'* or *B'*, is active. When the enabled transition fires, the nested state associated with the currently active shadow state is entered. As a result of firing *T11*, state *I3* is deactivated; composite state *X* and two of its nested states are activated.

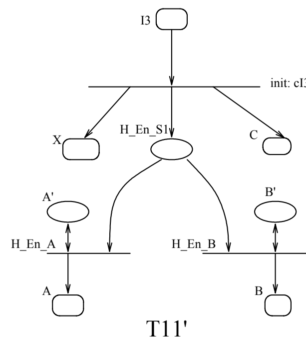
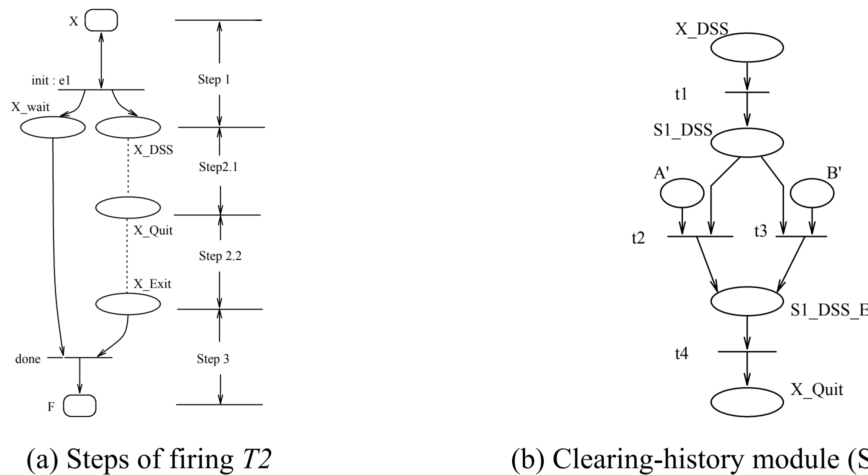


Fig. 22 Translation of entry transition *T11* in Fig.22

One important issue is that we must clear the old history by deactivating the shadow states before the new history can be recorded. We define a reusable module for clearing the history. Note that after history entry, the shadow states are still active. As shown in Fig. 22, the arcs connected to the shadow

states are bi-directional arcs. The reason is to allow the “clearing-history” module to have a generic form so it can be reused. The action of clearing shadow states can be performed in the process of firing exit transitions. More specifically, the shadow states are deactivated before the source states of exit transitions are deactivated so that the currently active nested states can be remembered by the shadow states when the nested states are deactivated. In light of this, the second step of modeling exit transitions is divided into two sub steps, Step 2.1 and Step 2.2. Step 2.2 is the same as the second step defined previously in Section 4, while Step 2.1 is a new step added to deactivate shadow states. The idea is that a control-state, *X_DSS* (In state X, Deactivating Shadow States), is added to drive the deactivation of shadow states. State *X_DSS* activates other control-states, as many as one for each orthogonal region that has a history state. The activeness of these control-states leads to the deactivation of the currently active shadow states.

Let us illustrate the idea via our running example. Consider transition *T2* of the statechart in Fig. 20. As shown in Fig. 23 (a), the model for transition *T2* has a refined step 2 that takes into consideration the deactivation of the shadow states. When transition *init* fires, state *X_DSS* becomes active and Step 2.1 starts. The details of Step 2.1, which deactivates the shadow states, are shown in Fig. 23 (b); this presents a “clearing-history” module. In general, such a module will include one transition associated with each shadow state, for the purpose of deactivating that shadow state. In our example, the firing of transition *t1* activates control-state *S1_DSS*. Then one of the two transitions *t2* or *t3* is enabled depending on which shadow state, *A’* or *B’*, is active. When the enabled transition fires, the currently active shadow state in region *S1* is deactivated, and control-state *S1_DSS_E* (In Region S1, Deactivating Shadow State End) becomes active. The firing of transition *t4* concludes the step of deactivation of shadow states and starts Step 2.2 with the activation of control-state *X_Quit*.



(a) Steps of firing *T2* (b) Clearing-history module (Step 2.1)

Fig. 23 Translation of exit transition *T2* in Fig.22

Another issue for consideration is the initialization of shadow states. More specifically, upon initial entry to the state machine, which shadow states are active? Regions that have a history state require the specification of a default history state, as we defined previously. Thus, shadow states associated with the

default history states should be active at the initialization of the state machine. The simplest approach to ensure this property is to include in the target Petri net's initial marking those place nodes that correspond to the shadow states associated with the default history states.

We now present the semantic rule for dealing with history states. In addition, the semantic rule for cross-boundary entry transitions (Rule 3.4) is refined to Rule 3.4' in order to include the case of history entry. In light of the need for clearing-history modules in dealing with history states, we present a slightly revised version of the translation rule, Rule 4.1, presented previously.

Semantic Rule 5.1: History Entry. Given a concurrent composite state *X* with a set of regions *R* and some region $r_i \in R$ has a shallow history state: If r_i is entered through history entry, then $\exists$ *S',* a shadow state belonging to region r_i, such that *S'* is active, and *S* is entered, where *S* is the state associated with the shadow state *S'*.

Semantic Rule 3.4': Cross-Boundary Entry Transitions. Given a concurrent composite state *X* with a set of regions *R* and an entry transition *t* that targets nested state *A* belonging to region $r_i \in R$: If *t* fires, then 1) state *X* is entered; 2) region r_i is entered explicitly (Rule 3.2 or Rule 3.2') if state *A* is not a history state, or through history entry (Rule 5.1) if state *A* is a history state; and 3) $\forall$ $r_j \in R$ such that $r_j \neq r_i$, r_j is entered by default (Rule 3.1).

Translation Rule 4.1': Extended Structure for Exit Transitions. Given a concurrent composite state *X* with a set of regions *R* and an exit transition *t*: The model for transition *t* consists of an *initial* transition for recognizing that transition *t* is enabled, a clearing-history module (if history states are present) for deactivating the shadow states, a *deactivation* module for deactivating the source states, and an *activation* transition for activating the target state.

To illustrate Rule 4.1', we apply this rule to exit transitions *T1*, *T3*, and *T4*, obtaining the models shown in Fig. 24. Note that each exit transition model consists of two unique transitions, *init* and *done*, and two shared modules, module 1 and module 2. Transition *init* represents the *initialization* transition that initiates the exit transition, module 1 represents the *clearing-history* module, module 2 represents the *deactivation* module, and transition *done* represents the *activation* transition that activates the target state. Module 1 is detailed in Fig. 23 (b), and module 2 is detailed in Fig. 21.

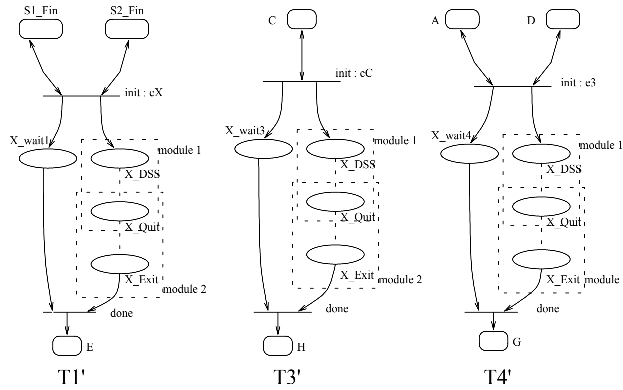


Fig. 24 Translation of exit transitions *T1*, *T3* and *T4* in Fig. 20

6 Completion-event semantics

Entry and exit transitions can be completion transitions. We call such transitions *completion-entry transitions* and *completion-exit transitions*. These transitions have associated with them both entry/exit semantics and completion-event semantics. In Sections 3.3 and 4, we discussed how to model entry and exit semantics. Recall that the key components of the generated models are the source configuration (for entry transition models), the direct source states (for optimized exit transition models), an *init* transition for recognizing the enabling condition of the entry/exit transition; and a control flow that leads to the target configuration. Fig. 25 (a) and (b) show an abstract architecture for entry and exit transition models, respectively. We now discuss how to model completion-event semantics by modifying the control flows for entry and exit transitions.

We first briefly review completion-event semantics as defined in UML [OMG03]. When all tasks (transitions, entry actions and activities) within the source state are complete, a completion event instance is generated[6]. In particular, if the source state is a composite state, a completion event instance is generated when the control flow for each orthogonal region of the composite state reaches the final state. The completion event then becomes the implicit trigger for the completion transition.

To model completion-event semantics the model must make explicit the implicit semantics associated with a completion transition. In other words, we must handle both features – the generation and recognition of the completion event. In previous sections, we simply represented completion events by assuming implicit triggers, which we modeled explicitly as completion triggers. The completion trigger is recognized by the *init* transitions, shown in Fig. 25 (a) and (b). Thus, since a completion trigger models a completion event, the completion event recognition problem is handled.

We now show how to model the generation of completion events. For a completion-entry transition, the architecture shown in Fig. 25 (a) needs to be refined. The new architecture is shown in Fig. 26 (a). In

[6] For convenience, we assume here that source states of completion transitions do not specify any internal transitions, entry actions, or activities.

the new architecture, a new transition, *CE_creation*, is introduced to explicitly generate the completion event. Details on how the completion event is modeled and stored in the event queue are described in Section 8, which discusses the issue of event dispatching.

For a completion-exit transition, we refine the model of Fig. 25 (b) into Fig. 26 (b). Again, transition *CE_creation* is introduced to generate the completion event. Note that the source configuration is preserved after transition *CE_creation* fires by the arrow pointing to the direct source state(s). This is to be consistent with the optimized approach for translation of exit transitions, for which the source configuration needs to be preserved even after the firing of the *init* transition. Control-state *lock* is introduced to ensure that exactly one completion event is generated per firing of the exit transition. Initially, control-state *lock* is active.

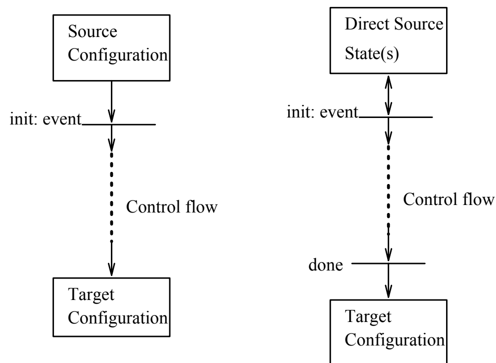


Fig. 25 Abstract architecture for entry and exit transition models

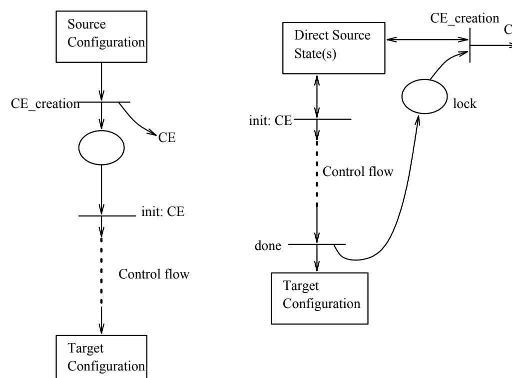


Fig. 26 Abstract architecture for completion-entry and completion-exit transition models

For illustration, we now re-consider the translation of two completion transitions in our running example of Fig. 8. *T7* is a completion-entry transition while *T1* is a completion-exit transition. We discussed how to partially translate these two transitions in Sections 3.3 and 4, respectively, ignoring the completion-event semantics. Since both transitions are completion transitions, a completion event instance needs to be generated for each transition. Based on the modeling technique just defined, the net model for transition *T7* is modified (and reduced by Rule 3.5) and shown in Fig. 27. Similarly, the refined model for completion-exit transition *T1* (previously shown in Fig. 17 (a)) is shown in Fig. 28.

We note that a UML completion transition that is a simple transition can be modeled in a way that is similar to a completion-entry transition. For the simple transition, the target state, which becomes active after recognition of the completion event, is a single simple state.

According to UML semantics, completion events themselves are events that must be dispatched. They play an important role in the process of event dispatching and help define the run-to-completion step, which is described in Section 8.

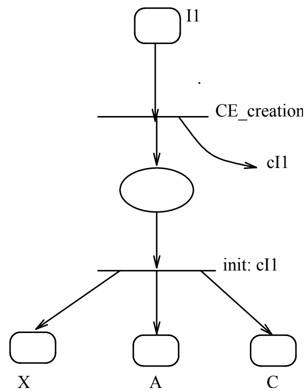


Fig. 27 Modified net model for transition *T7*

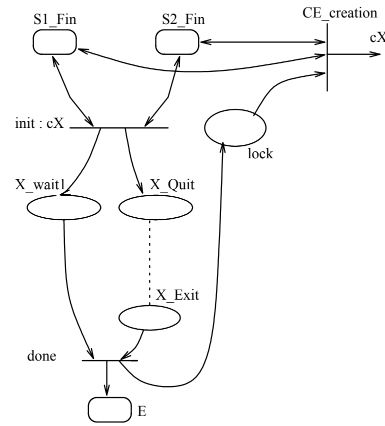


Fig. 28 Modified net model for transition *T1*

7 Some Modeling Implications

7.1 Mapping Analysis Results Back to UML Notation

In order to map analysis results at the net level back to UML notation, one issue that must be addressed is establishing correspondence between the UML transition and the behavior of the associated ISM model – in other words, how to recognize the UML transition in the obtained model so that the information about the UML transition can be recorded in simulation traces when the corresponding net transitions fire. More specifically, there are two key issues. We will first discuss these two issues in terms of exit transitions.

First, *what* information should be recorded during model execution? To allow the net behavior to be interpreted in terms of components of the UML transition, we should record the *source composite-state*; which will be the composite state being exited; triggering event; and target state. The second key issue is *when* the information should be recorded. In other words, what transitions in the net structure need to be made "visible" in terms of contributing information during model analysis or simulation? The source state and the triggering event can be obtained from the initialization transition, *init*. For example, in Fig. 17 (a), the transition *init* targets the control-state *X_wait*, which indicates that the source composite-state for the corresponding UML transition (*T2*) is composite state *X*. Similarly, the trigger for transition *init* is event *e1*, which indicates that the triggering event for UML transition *T2* is also event *e1*. The target state can be identified from the final transition of the net structure, the so-called *done* transition. For example, in Fig. 17 (a), the target state for transition *done* is state *F*, which is also the target state of the UML transition. Thus, the critical transitions in the net structure are the *init* and *done* transitions since these two transitions are associated with the information regarding the UML transition. The firings of these two transitions symbolize the behavior defined for the UML exit transition. The other transitions associated with the net structure, although they do fire in some order, can be made "invisible" (so-called "silent transitions") for the purpose of mapping our model's behavior back to the UML notation.

Entry transitions can be treated similarly. The source state and triggering event of an entry transition can be easily identified from the *init* transition in the net model for the entry transition. In cases that do not involve history states, the target configuration can also be identified directly from the *init* transition. Recall that the model for an entry transition that targets a history state is a 2-level model. The *init* transition is followed by a set of 2nd-level transitions. In the case of such an entry transition, the *init* transition indicates the source state, the triggering event, the composite state being entered, any target states that are entered by default, and the history state. The 2nd-level transitions indicate which particular nested state is entered via history entry.

As a small example, consider our running example in Fig. 8. Assume that the object leaves state *I1* and enters composite state *X* by entry transition *T7*, and subsequently event *e1* occurs, causing the object to leave composite state *X* and enter state *F* by exit transition *T2*. Based on our modeling approach and the mapping ideas described above, the trace information that would be collected is the following:

1. Transition *T7* fires. Source state: *I1*; Trigger: no trigger; Target Configuration: *X, A, C* /* Due to the *init* transition for entry Transition *T7*; See Fig. (9) (and the control-state reduction rule) */
2. Transition *T2* begins. Composite state being exited: *X*; Trigger: *e1* /* Due to the *init* transition for exit transition *T2*; See Fig. 17(a)*/
3. Transition *T2* ends. Target State: *F* /* Due to the *done* transition for exit transition *T2;* See Fig. 17(a) */

Note that many of the intermediate net-level transitions are not visible by the above mapping. Note also that transition *T2* consists of two separate stages. Here, we focus on the idea of mapping the meaning of these two stages in terms of how the firings of the net model transitions can be mapped back to the meaning of the UML model. As an optimization, these two stages can be combined as one atomic step by using a technique similar to that described in the following section. For lack of space, the details on this matter are not presented in this paper.

7.2 Atomicity of UML Transitions

Note that the behavior of an exit transition does not automatically correspond to the firing of a single transition in the net model. We assume that the semantics of our net model allow the net transitions (visible or invisible) that form the net structure to interleave with net transitions corresponding to other concurrent UML transitions[7]. But since none of these invisible transitions is externally observable, such an interleaving does not impact the desired semantic behavior. Yet, there are some cases where such

[7] An exit transition can be concurrent with other transitions when the exit transition is enclosed in a region of a concurrent composite state.

interleaving must be prevented, such as when there are data-oriented actions associated with UML transitions. For instance, in the case of concurrent composite states, nested transitions that belong to different regions may be concurrently enabled and the firings of these transitions are allowed to interleave with each other. However, when actions such as read, write, or computation are associated with such transitions, the actions should not overlap, or be performed in parallel; our model must ensure a mutual exclusion semantics for such actions.

To model the atomicity of concurrent UML transitions, we use a standard net modeling technique based on introducing a control-state *lock* for each concurrent composite state. Control-state *lock* is connected to a set of net structures, each modeling one of the nested transitions[8]. Fig. 29 illustrates the basic idea. Fig. 29 (a) shows a concurrent composite state that has two concurrent transitions *t1* and *t2*. Both transitions have associated actions, *action1* and *action2*, respectively. Fig. 29 (b) shows how control-state *lock* is used to ensure the atomicity of firing the net models for transitions *t1* and *t2*. Since the modeling of specific actions (in particular, reads and writes of variables) is outside the scope of this paper, we simply use a dashed-box to denote an abstract model for such actions. We call these generic models *action-models*. Note that while any one net structure is active, control-state *lock* is inactive. Thus no other net structures can be enabled. Therefore the execution of any action-model is atomic. Control-state *lock* should be set as active at the initialization of the state machine.

Taking this a step further, we can realize that a system model can contain several statecharts, so a UML transition may interleave also with transitions of other statecharts. If these transitions are associated with actions that process common data (a global variable), the firing of these transitions should be atomic. This case can be modeled using the same basic approach as described above.

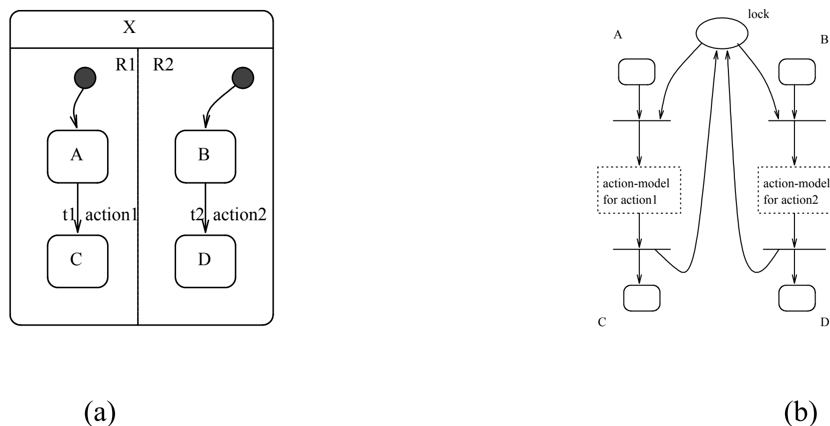


Fig. 29 An example for using control-state *lock* to ensure atomicity of UML transitions

[8] As an optimization, the connections (arcs) associated with the lock place need not be generated for a nested transition that does not involve (data-oriented) actions.

7.3 Transition Conflicts and Priorities

We now discuss another issue, which relates to exit transitions – transition conflicts and transition priorities. Consider Fig. 30. Note that since transitions *T1* and *T2* have the same triggering event *e*, they can be enabled simultaneously, for example, when states *A* and *C* are active and event *e* occurs. In general, *two transitions can be in conflict if they have the same triggering event and at least one source state in common.*

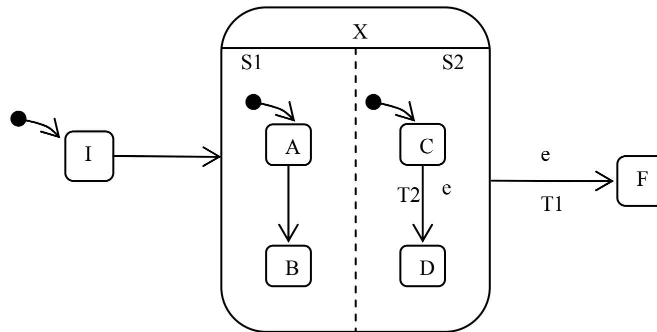


Fig. 30 An example statechart involving transition conflicts

It is sometimes possible to resolve conflicts with the help of the priority of transitions defined by the UML standard. UML semantics [OMG03] define firing priorities for situations where there are conflicting transitions. These priorities of conflicting transitions are based on their relative positions in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of the substate's containing states. More specifically, if *t1* is a transition whose source state is *s1*, and *t2* has source state *s2*, then: 1) If *s1* is a direct, or transitively nested, substate of *s2*, then *t1* has higher priority than *t2*; and 2) If *s1* and *s2* are not in the same state configuration, then there is no priority difference between *t1* and *t2*. For example, in the case of the example, transition *T2* has higher priority than *T1*.

The key issue of handling transition priority is to decide which transition can fire. Since firings of transitions are initiated by triggering events and transitions in conflict are triggered by the same event, the issue becomes which transition can consume the triggering event, i.e., which transition can "see" the triggering event first. In Petri nets, we can model the event dispatcher mechanism in such a way that an event will be dispatched to the transitions with higher priority first. If the event is not consumed by the transitions with higher priority, it will be dispatched to those with lower priority. In Section 8, we will show how the event dispatcher mechanism is modeled. Gabor and Istvan implemented this idea with Stochastic Reward Nets (SRN) in [GI00]. One apparent advantage associated with this approach is that we need not be concerned about the issue of transition priority when we derive ISMs from composite states. Therefore, it simplifies this part of the transformation.

Sometimes, conflicts cannot be resolved by priorities of transitions. Suppose that in Fig. 8 transition *T3* is triggered by event *e2*. Then *T6* and *T3* would be in conflict given that state *C* is active. But transitions *T6* and *T3* have no priority over each other since they have the same direct source state, (in this case, the state *C*). In this situation, we ignore the conflict and the selection of which transition to fire can be modeled as a non-deterministic choice.

8 Event Dispatching

In previous sections, we use a state-transition notation to make explicit the control flow of a state machine when an event is dispatched. Now, we discuss another aspect of UML state machine semantics, how to model event dispatching. Events must be made available to drive the control flow of the state machine. To explain the details on modeling the event dispatcher mechanism, we now view the target models in terms of colored Petri net models.

According to UML state machine semantics [OMG03], events are dispatched and processed by the state machine one at a time. More specifically, the semantics on event dispatching and processing are based on the following key features.

1. A state machine has *an event queue*, which holds incoming event instances until they are dispatched.
2. The semantics of event processing are based on the *run-to-completion processing*, which means that an event can only be dequeued and dispatched if the processing of the previous current event is fully complete. Furthermore, if a dispatched event does not trigger any transition, then the event is "discarded."
3. In the presence of concurrent composite states, it is possible to *fire multiple transitions* as a result of the same event – as many as one transition in each concurrent state in the current state configuration.
4. *A completion transition* is triggered implicitly by a *completion event*, which is generated when all transitions and entry actions and activities in the currently active state are completed. The completion event is dispatched before any other queued events.
5. In situations where there are *conflicting transitions*, the selection of which transitions will fire is based in part on a priority. By definition, a transition originating from a substate *S* has higher priority than a conflicting transition originating from any states that contain state *S*. For composite states that have only one level of nesting, the issue is that boundary exit transitions should have lower priority than nested transitions and cross-boundary exit transitions. For simplicity of presentation, this is the case that we consider in this section.

Based on the semantics for event dispatching, we design an abstract net model for handling event-dispatching issues. To begin, we present in Fig. 31 a net model sketch that addresses the following three issues: 1) how an event-token in the event queue is selected and dispatched; 2) how the dispatched event-token, also called the *current* event-token, is made available to net transitions that may or may not be associated with composite states; and 3) how an event-token for a completion event is generated and deposited into the event queue.

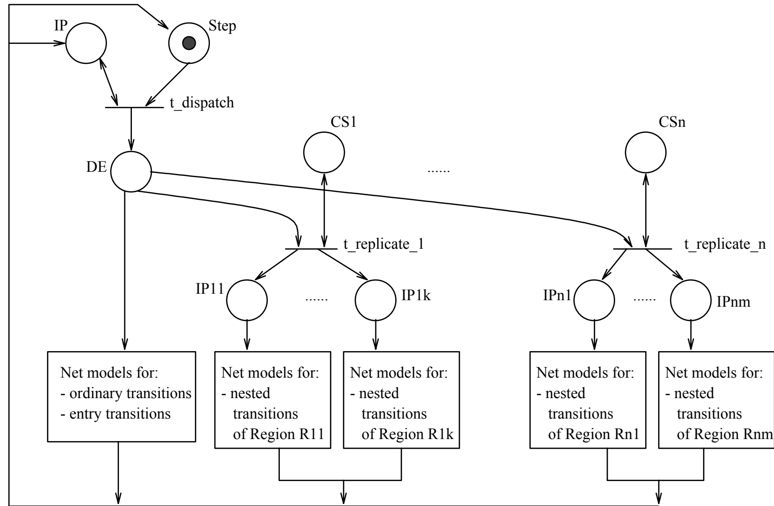


Fig. 31 An event-dispatching model (incomplete)

To address the first issue, an event-token in the event queue should be allowed to be dispatched only if the previous *current* event-token is fully processed. To model this, we introduce a place, *Step* in Fig. 31, to control the dispatching, i.e., to initiate a run-to-completion step. The actual dispatching involves the selection of an event-token in the event queue. Since it is common to assume a FIFO property on the event queue, we must be specific on the way that event-tokens are held in the *IP* place. In our model, the event queue is modeled by a colored token held within the *IP* place, rather than by the *IP* place itself. This special token is called the *event-queue-token*. As we know, the "color" of a token is a kind of data type. We use a list as the data type for the event-queue-token, and event-tokens are elements of this list. Thus, when an event-token needs to be dispatched, the first event-token in the list must be selected. The *t_dispatch* transition removes the event-queue-token from place *IP*, extracts the "first" event-token from the list, and deposits the extracted event-token into place *DE*. Furthermore, *t_dispatch* must redeposit the updated event-queue-token back into place *IP*. For simplicity we do not show the token colors and enabling conditions for transitions in our net model sketch.

To handle the second issue – making the dispatched event-token available to net transitions – place *DE* connects to two classes of net transitions: 1) Those transitions that model UML simple transitions that are not contained within a composite state, which we refer to as *ordinary transitions*; and the *init* transitions associated with models of UML entry transitions; and 2) Those transitions that model nested

transitions associated with composite states. An event-token can be allowed to directly trigger a transition of the first class since no active composite state is involved. Yet, for the second class, a composite state is active. So, based on feature 3, the current event-token needs to be allowed to trigger multiple transitions, at most one for each concurrent region. To model this case, we replicate the current event-token so that each region has a copy of this event-token. As shown in Fig. 31, when composite state *CS1* is active, the current event-token is replicated via the firing of transition *t_replicate_1*. Places *IP11…IP1k* are defined to hold the copies of the current event-token for each region.

As discussed in Section 6, modeling of completion-event semantics involves generation of completion events. Recall (from Section 6) that in our net model a *completion-event-token* is generated by a "*CE_creation*" transition. To preserve the semantics implied by feature 4, a *CE_creation* transition should have priority over the *t_dispatch* transition. Due to space considerations, we do not discuss further this net-level modeling detail. A generated completion-event-token is put in the front of the event queue list, unlike other generated event-tokens that by default go to the end of the event queue list. Therefore, in the next run-to-completion step, the completion-event-token will be dispatched first, which will then trigger the associated completion transition model.

Now consider the issue of transition conflicts and priority. As mentioned in Section 7.3, there are two cases for transition conflicts: 1) transitions in conflict have the same priority; and 2) transitions in conflict have different priorities. Case one can be modeled as a non-deterministic choice. The idea is making an event-token available to multiple transitions. Although more than one transition (that are in conflict) can be enabled, only one of the enabled transitions is able to fire. The choice is non-deterministic. A special case of this kind is the conflict between a cross-boundary exit transition and either a nested transition or another cross-boundary exit transition. In particular, a cross-boundary exit transition associated with one region can be in conflict with some transitions associated with another region. If such conflict exists, only one of these transitions should fire. To model this behavior in our ISM model, all the copies of the event-token are used to enable each *init* transition defined in the ISM models for cross-boundary exit transitions. Note that cross-boundary exit transitions belong to the second class of transitions as discussed earlier since cross-boundary exit transitions are associated with active composite states.

The second case of transition conflict is specific to situations that involve composite states. For such situations, we should ensure that nested transitions and cross-boundary exit transitions have higher priority than boundary exit transitions. We can model this behavior by making the current event-token available first to nested transitions and cross-boundary exit transitions. If such transitions do not consume the current event-token, the token will then be available to boundary exit transitions. To explain this further, we extend the event dispatching model of Fig. 31.

Without loss of generality, we illustrate the key ideas for handling event dispatching, in particular the issue of transitions in conflict having different priorities, via the case of a statechart that contains one composite state with two concurrent regions, *S1* and *S2*. Fig. 32 illustrates the basic structure of the new event dispatching model for a system that includes such a case.

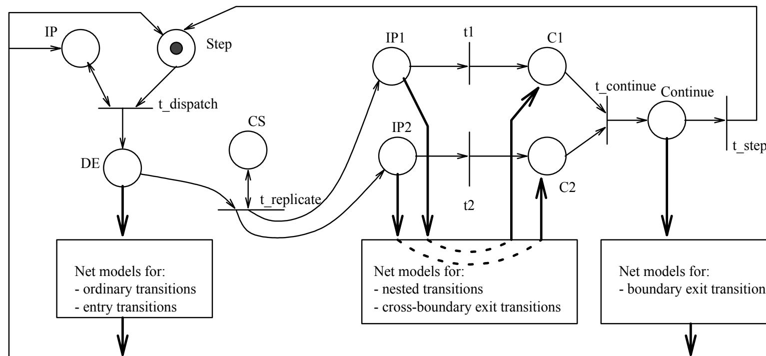


Fig. 32 The event dispatching model (with priority/conflict handling)

In Fig. 32, places *IP1* and *IP2* are connected to the net models for nested transitions and cross-boundary exit transitions as mentioned earlier, while place *Continue* is used to make an event-token available to boundary exit transitions or to initiate the next run-to-completion step. If the event-token in place *IP1* is consumed by the firing of a net transition corresponding to a nested or cross-boundary exit transition associated with region *S1* (call it transition *t*), a token with label "consumed" will be deposited into place *C1* (by the firing of transition *t*). This indicates the completion of the current step for region *S1*. Otherwise, if the event-token does not trigger any such transition, the current event-token will be deposited into place *C1* by the firing of transition *t1*. Similarly, place *C2* will hold either a consumed-token or the current event-token depending on whether or not the current event-token triggers a transition associated with region *S2*. Note that transition *t1* is enabled when the event-token cannot enable any nested transition model for region *S1* or any cross-boundary exit transition model. Transition *t2* is enabled similarly. The specific arc-inscription labels for these enabling conditions are not shown here to simplify the figure. Assume that both places *C1* and *C2* hold a current event-token. When transition *t_continue* fires the current event-token will be deposited into place *Continue*. Thus the event-token is available to enable any boundary exit transition models associated with the composite state. Otherwise, if at least one of the two places *C1* or *C2* holds a consumed-token, a consumed-token will be deposited into place *Continue* by the firing of transition *t_continue*. In this case, the event-token is not available to enable any boundary exit transition models. Finally, transition *t_step* is defined to initiate the next run-to-completion step. When transition *t_step* fires, place *Step* is marked and a new run-to-completion step can start.

To summarize, each run-to-completion step is completed when the current event-token (in place *DE*) is "fully processed," meaning that one of the following conditions is satisfied: 1) The current event-token triggers an ordinary transition or an entry transition; 2) The current event-token triggers any nested transition or a cross-boundary exit transition – place *Continue* holds a consumed-token that triggers transition *t_step*; 3) The current event-token triggers a boundary exit transition; or 4) The current event-token cannot trigger any transitions of the state machine – in effect, the event-token is "discarded" by transition *t_step*. When the current event-token is fully processed, place *Step* will again hold a token, resulting in a new event-token being dispatched.

Now we can present a semi-complete model for the running example of Fig. 8, by instantiating the template model of Fig. 32. The resulting model is shown in Fig. 33. To maintain readability of the figure, the net structures for some transitions (*T4*, *T6*, *T8*, and *T10*) are not shown – they are similar to those that are shown. Also, some places are replicated for clarity[9]. The top of the figure shows the event dispatching model. This model connects to the net model of the state machine as follows. Place *DE* is connected to the net transition (*T7_init*) that models the entry transition, transition *T7*. As a labeling convention, we prefix net transition names with the corresponding UML transition name. Place *IP1* is connected to two net models for the two nested transitions of region *S1*, *T5* and *T9*. Both places *IP1* and *IP2* are connected to the net model for the cross-boundary exit transition *T4*. Place *Continue* is connected to the net models for the two exit boundary transitions, *T1* and *T2*. Note that in Fig. 33 there exist *CE_creation* transitions for completion events *cI1, cB,* and *cX*. The arc inscription "*cI1+eqt*" denotes that the completion-event-token *cI1* is appended to the front of the event-queue-token. The other completion-event-tokens are handled similarly. The details of the deactivation module shown in Fig. 33 were presented earlier in Fig. 17 (b). Moreover, in the cases that the current event-token triggers an entry transition or a boundary exit transition, Fig. 33 shows how place *Step* is marked to indicate that a run-to-completion step is finished. The other case that defines a run-to-completion step is that the token in place *Continue* cannot enable any boundary exit transition model. As discussed previously, this case is taken care of by transition *t_step*. Note that the enabling condition for *t_step* specifies that the token in place *Continue* cannot enable any boundary transition model because 1) the current token is a consumed-token (as discussed previously), or 2) the token is an event-token but it is neither *cX* nor *e1*.

[9] A replicated place is named by attaching "*" to the original name. For example, a place named as *IP** represents a copy of place *IP*.

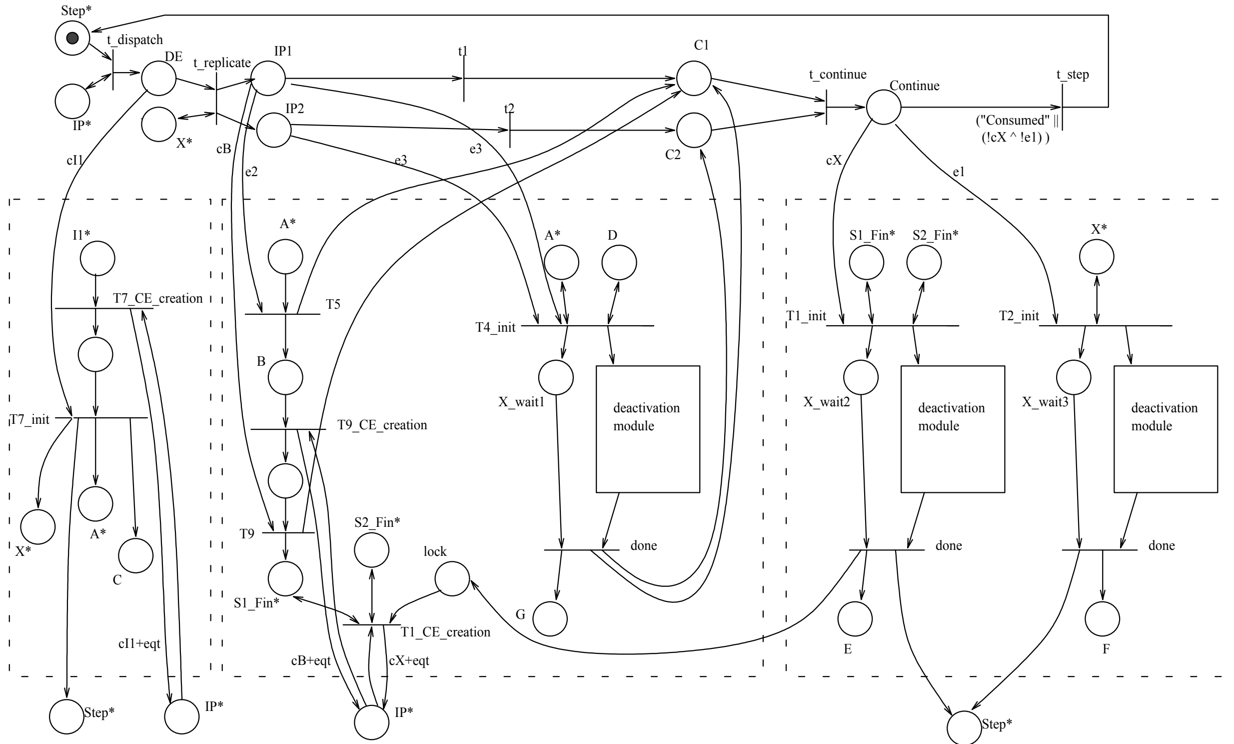


Fig. 33 The event dispatching and processing model for the running example of Fig. 8

9 Related Work

Applying Petri net modeling to UML statecharts is a currently active research area. Other works in this area include the work of Dong, et al. [DFH03] and Merseguer [Mer03]. In [DFH03], they convert UML state machines into a type of Petri net called Hierarchical Predicate Transition Net (HPrTN) and realize implied mechanisms in UML state machines. In [Mer03], a formalization for a subset of UML statecharts is proposed in terms of another type of Petri net call Generalized Stochastic Petri Net (GSPN). These works are motivated by concerns very similar to our own but have different strengths. In [DFH03], an approach is presented to define a complete formal semantics of UML state machines based on HPrTNs. In [Mer03], a formal translation from UML state machines to Petri nets is presented for the purpose of studying performance of software systems. However, our primary focus is to develop a process to build a state-transition model that facilitates the translation from UML statecharts to colored Petri nets. One advantage of our approach is its design strategy of separation of concerns. Thus, each involved issue can be addressed more thoroughly. In [DFH03], they define the priority levels for different types of transitions but they do not actually show how to implement transition priority in the net model. In [Mer03], cross-boundary transitions - transitions that target to, or originate from, the nested states of composite states – are not discussed. Moreover, history states are only briefly addressed. In our work, we

discuss an approach for implementing transition priority in the target model. In addition, we present approaches of modeling cross-boundary transitions and history states.

There exists previous related research that focuses on formally defining the semantics associated with composite states. In [LP99], Lilius and Paltor presented a formalization of UML state machine semantics. This formalization has been used as the basis for developing the vUML tool, a tool for model checking UML models [PL99]. In that work, it is suggested that the hierarchical structure must be "flattened" before using it in a model checker. We agree with this opinion in the sense that in our framework the hierarchical structure must be expanded before generating CPN models. An issue for consideration relates to their method to interpret the semantics of transitions to and from composite states. They interpreted the semantics using a textual notation for states, which makes the hierarchical relationship of states explicit. However, in their interpretation, they did not expand the transitions completely. Instead, they used state variables to allow an expression for one transition to represent multiple transitions. This technique is not sufficient for our purpose since we aim to generate Petri net models where each transition needs to have specific source and target states. To address this issue, we discuss an expansion method. We recognize that this type of expansion is prone to the so-called state explosion problem since the number of resulting transitions increases dramatically with the increasing of the number of orthogonal regions and the number of nested states within an orthogonal region. To cope with this problem, we propose optimization methods to decrease the number of resulting transitions. In our presentation for state hierarchy, we have adapted Lilius and Paltor's notation, but state variables are not used.

In [LBE00], Lano et. al. provided a systematic formal interpretation (extended first-order set theory) for most elements of the UML notation. In terms of statecharts, some reductive transformations are used to eliminate sequential composite states, concurrent composite states, and entry and exit actions. With their approach, a concurrent state is expanded to a state machine for which states are tuples. A tuple consists of basic states, each from a different orthogonal region. This approach presents explicit semantics for transitions to and from composite states as well as a method for translating entry and exit actions associated with composite states. However, this approach does not address issues regarding transition priority and history states, which interfere with hierarchy and complicate the task of transforming UML statecharts to formal notations. In [Bin00], an approach is proposed to expand the statecharts' implicit state machine to generate a complete test suite. The orthogonal regions of a concurrent composite state are translated into a single product machine. In contrast with [LBE00] and [Bin00], our approach preserves the feature of concurrency in statecharts. Moreover, the state explosion problem associated with expanding concurrent states is not addressed in either [LBE00] or [Bin00]. We described an approach for decreasing the number of resulting transitions to help mitigate the state explosion problem, as presented in Section 4.

Some other works that involve expanding statecharts are as follows. Latella et. al. proposed to use *Extended Hierarchical Automata* (EHA) as an intermediate model for model checking UML statecharts [LMM99]. Source restriction and target determination are used to help specify the source/target states of transitions from/to composite states in EHA. The main focus of that work is to define a formal model for UML state machines in the form of extended hierarchical automata. However, the translation of UML statecharts to the intermediate model is only briefly outlined. In contrast, our work presents a systematic process for building an Intermediate State Machine model, which makes explicit a subset of UML statechart semantics. In [GP98], Gogolla and Presicce proposed to transform UML state diagrams into graphs by making explicit the intended semantics of the diagram. Their purpose is similar to that of our work; however, their discussion focused on the sequential composite states. In contrast, our work focuses on the complex issues involved in concurrent composite states.

10 Conclusions and Future Work

Previously, we defined a general Petri-net based methodology for statechart formalization and analysis. A key advantage of the approach is that it can exploit existing Petri net theory and tools to support automated analysis of UML specifications. To extend the approach to handle composite states, this paper addresses the problem of explicitly modeling the semantics of the complex transitions associated with composite states. More specifically, we define a process for deriving a state-transition model that covers all of the primary issues associated with the hierarchical structure of composite states. This model serves as an intermediate, platform independent model that can be mapped to a particular colored Petri net notation (CPN). In this sense, the technique conforms to the Model Driven Architecture methodology. In other words, the intermediate model delays binding our transformation approach to a specific CPN notation so that our transformation approach can be implemented on top of a standard CPN analyzer to support model analysis and simulation.

We discussed in detail the specific tasks necessary for composite state handling: entry and exit transitions, transition conflict and priority, and history states. Moreover, we presented a translation optimization for exit transitions to reduce the size of the obtained model. Since our model has a generic form, it naturally supports the automation for the translation process. Furthermore, we presented an abstract net model for handling the semantics associated with the event dispatcher mechanism of UML state machines.

One direction of future work is to extend our current tool support. Currently we have a prototype tool that supports automatic translation of basic UML statecharts, which do not contain composite states, into colored Petri nets. Naturally, the next direction is to integrate the ability to handle composite states into the tool. More specifically, two issues are involved. First, we will automate the transformation process for converting UML composite states into our intermediate state-transition model. Furthermore,

we plan to define and automate a mapping from the intermediate state-transition model to a "platform specific" CPN model, such as Design/CPN. This will allow us to do further experimentation on automated analysis and simulation of UML diagrams.

Recall that we mentioned in Section 1.2 that some statecharts features are not handled in our current framework. Another direction for future research is to extend our framework to handle these features, such as guard conditions, deferred events, actions that involve variables, and activities. For instance, to handle guards, we should be able to take advantage of colored Petri net modeling, which also includes the concept of guards on transitions. So, statechart guards can be mapped into colored Petri net guards. These guards define conditions that are automatically checked in order to allow associated transitions to fire. Since guards in statecharts are typically defined using variables, we will also need to investigate methods for tracking the values of variables through the attributes of colored tokens. A Petri net transition or a set of net transitions can be used to model actions and activities. In our approach, an event queue is modeled by a list that was discussed in Section 8. Thus, this should be able to support the modeling of deferred events. In particular, since a list preserves the order of its elements, the events stored in a list can be examined one by one to check if any of them are deferred events.

Acknowledgment

We thank J. Saldhana and the reviewers for their helpful comments during the writing of this paper.

Reference

[Bee01] M. von der Beek. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, editors, *Proc. of UML´2001 – The Unified Modeling Language* , volume 2185 in LNCS. Springer Verlag, 2001.

[BC+01] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Fataricza and G. Savoia. Dependability Analysis in the Early Phases of UML-based System Design. *Journal of Computer Systems Science and Engineering,* 16(5), 2001, pp. 265-275.

[Bin00] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison Wesley, 2000.

[BJR99] G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[BL+00] J. M. Bruel, J. Lilius, A. Moreira, and R.B. France. Defining Precise Semantics for UML. In J. Malefant, S. Moisan, and A. Moreira, editors, *ECOOP 2000 Workshop Reader*, volume 1964 in LNCS. Springer Verlag, 2000.

[BP01] L. Baresi and M. Pezzè, On Formalizing UML with High-Level Petri Nets, In G. Agha and F. De Cindio, editors, *Concurrent Object-Oriented Programming and Petri Nets* (a special volume in Petri Nets series), volume 2001 of LNC.Springer Verlag, 2001, pp. 271-300.

[CHS00] K. Compton, J. Huggins, and W. Shen. A Semantic Model for the State Machine in the Unified Modeling Language. In S. Kent , A. Evans, and B. Selic, editors, *Proc. of UML'2000 Workshop - Dynamic Behaviour in UML Models: Semantic Questions,* volume 1939 in LNCS. Springer Verlag, 2000.

[DCPN] Design/CPN, Available at www.daimi.au.dk/designCPN/.

[DFH03] Z. Dong, Y. Fu, and X. He. Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams. In *Proc. of Software Engineering and Knowledge Engineering,* 2003.

[FE+98] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In *Proc. of UML'98 Workshop – Beyond the notation*, volume 1618 of LNCS. Springer, 1999, pp. 336-348.

[GI00] H. Gábor and M. István. Quantitative Analysis of Dependability Critical Systems Based on UML Statechart Models. In *Proc. of Fifth IEEE International Symposium on High Assurance Systems Engineering*, 2000, pp. 83-92.

[GP98] M. Gogolla and F. Parisi-Presicce. State Diagrams in UML: A Formal Semantics Using Graph Transformations. In M. Broy, D. Coleman, T.S. E. Maibaum, and B. Rumpe, editors, *Proc. of PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998, pp. 55-72.

[GL+00] S. Gnesi, D. Latella, I. Majzik, and M. Massink. Formal Validation of UML Statechart Diagrams Models. In S. Kent , A. Evans, and B. Selic, editors, *Proc. of UML´2000 Workshop - Dynamic Behaviour in UML Models: Semantic Questions,* volume 1939 in LNCS. Springer Verlag, 2000.

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

[HS] Z. Hu and S. Shatz. Mapping UML Diagrams to a Petri Net Notation to Exploit Tool Support for System Simulation. In *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE'04).* 2004, pp. 213-219.

[KCJ98] L.M. Kristensen, S. Christensen, K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 1998, Springer Verlag, pp. 98-132.

[KP99] P. King and R. Pooley. Using UML to Derive Stochastic Petri Net to the Models. In *Proc. of the 15th Annual UK Performance Engineering Workshop*, 1999, pp. 45-56.

[Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In M. Gogolla and C. Kobryn, editors, *Proc. of UML'2001 – The Unified Modeling Language* , volume 2185 in LNCS. Springer Verlag, 2001.

[LBE00] K. Lano, J. Bicarregui, and A. Evans. Structured Axiomatic Semantics for UML Models. In *Proc. of ROOM 2000, Electronic Workshops in Computer Science*. Springer Verlag, 2000.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart diagrams. In *Proc. of FMOODS'99, IFIP TC6/WG6.1, Third IFIP International Conference On Formal Methods for Open Object-based Distributed System*, 1999, pp. 331-347.

[LP99] J. Lilius and I. Paltor. The Semantics of UML State Machines. Technical Report 273, Turku Centre for Computer Science, 1999.

[MC99] W. E. McUmber and B. H. Cheng. UML-based Analysis of Embedded Systems Using a Mapping to VHDL. In *Proc. of IEEE high Assurance Software Engineering (HASE99).* IEEE. 1999.

[Mer03] Jose Merseguer. *Software Performance Modeling Based on UML and Petri nets*. Doctoral Thesis, Universidad de Zaragoza, 2003.

[MSC99] ITU-T Recommendation Z.120: Message Sequence Chart. International Telecommunication Union; Telecommunication Standardization Section (ITU-T), 1999.

[Mur89] T. Murata, "Petri Nets: Properties Analysis and Applications," In *Proc. of the IEEE*, 77(4), April 1989, pp. 541-580.

[OMG03] OMG. UML semantics 1.5, 2003, available at www.uml.org.

[PG00] R. G. Pettit IV and H. Gomaa. Validation of Dynamic Behavior in UML Using Colored Petri Nets. In S. Kent , A. Evans, and B. Selic, editors, *Proc. of UML´2000 Workshop - Dynamic Behaviour in UML Models: Semantic Questions,* volume 1939 in LNCS. Springer Verlag, 2000.

[PL99] I. Paltor and J. Lilius. Formalizing UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language. Beyond the Standard*, volume 1723 in LNCS. Springer, 1999.

[Poo01] J.D. Poole. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *Proc. of ECOOP'2001Workshop on Object-Oriented Architectural Evolution*, 2001.

[PUML] The Precise UML Group. Available at www.cs.york.ac.uk/puml/.

[Reg02] G.Reggio. Metamodelling Behavioral Aspects: the Case of the UML State Machines Complete Version). Technical Report of DISI - Università di Genova, DISI-TR-02-3, Italy, 2002.

[Rum98] B. Rumpe. A Note on Semantics (with an Emphasis on UML). In Haim Kilov, Bernhard Rumpe, editors, *Proc. of Second ECOOP Workshop on Precise Behavioral Semantics,* Technische Universit"at M"unchen, TUM-I9813, 1998.

[SSH01] J. A. Saldhana, S. M. Shatz, and Z. Hu. Formalization of Object Behavior and Interactions From UML Models. *International Journal of Software Engineering and Knowledge Engineering,* 11(6), 2001, pp. 643-673.