Quantitative Analysis of Dependability Critical Systems Based on UML Statechart Models

Huszerl Gábor, Majzik István*
Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
H-1521, Budapest, Pázmány P. sétány 1/d., Hungary
[huszerl, majzik]@mit.bme.hu

Abstract

The paper introduces a method which allows quantitative performance and dependability analysis of systems modeled by using UML statechart diagrams. The analysis is performed by transforming the UML model to Stochastic Reward Nets (SRN). A large subset of statechart model elements is supported including event processing, state hierarchy and transition priorities. The transformation is presented by a set of SRN design patterns. Performance measures can be directly derived using SRN tools, while dependability analysis requires explicit modeling of erroneous states and faulty behavior.

1 Introduction

As the complexity of the computer systems used in our everyday life increases, the task of the engineers in developing these systems becomes increasingly difficult. Well-specified, easy-to-use environments and standardized design languages are required that support specification, design, verification and validation of complex (distributed) systems. A wide variety of languages and methods is available nowadays, however, the selection among them is not easy. On one hand, languages and formalisms that are amenable to (formal) analysis are usually not easy to understand and use by designers, since they require theoretical background and experience. On the other hand, widely-used design environments often lack of analysis capabilities. One way to bridge this gap is the development of methods to support the automatic analysis of models prepared using popular design languages.

The Unified Modeling Language (UML) [14] provides a standard visual notation for expressing the artifacts of complex distributed systems. It is the leading industry-standard language for object-oriented modeling and design of systems ranging from embedded systems to business applications. It is supported by a wide variety of tools and environments, offering services for specification, design refinement and automatic code generation. In the recent years, several methods were elaborated to support the analysis of UML based designs. Among others, problems of system-level dependability modeling [5], formal verification [10], performance analysis [7] of (subsets of) UML models were solved. [4]

Our work is focused on the quantitative analysis of the behavioral models of UML. The dynamic behavior of the system is described in UML by statechart diagrams [13], a variant of classical Harel statecharts [9]. They describe the internal behavior of components (objects, hardware nodes etc.) as well as their reactions to external events. The detailed description of the behavior by statecharts enables both quantitative performance analysis (as timing information is assigned to state transitions) and dependability analysis (in the case of extending the model with explicit failure states/events and probabilistic information). The standard UML notation has not been designed for and does not cover the aspects related to quantitative analysis, however, it provides standard mechanisms to extend the model both with timing/stochastic information (in the form of tagged values) and classification of model elements (in the form of stereotyped states and events).

In previous works [7] the quantitative analysis of a restricted subset of statecharts, the so-called Guarded Statecharts (GSC), was proposed. GSC models fit well for modeling of embedded systems where synchronization among components can be described solely by Boolean predicates on the active states of concurrent components. However, GSC models do not support event processing, which concept may be of crucial importance in modeling real dis-

*This work was partially supported by the "Hungarian-German Researchers Exchange Program" (DAAD-MÖB) project No. 8. and by projects OTKA-F030553, OTKA-T30804 and FKFP-0193/1999.

tributed systems. Moreover, it does not allow the use of state hierarchy, one of the most useful concepts in statecharts. In our work the input model is extended in special consideration of event processing and state hierarchy (the latter extension also comprising transition priorities).

The analysis, similarly to [7], is based on the transformation from UML statecharts to Petri net models with timing and stochastic extensions. Petri nets (PN) are a widely accepted formalism for modeling and analysis of distributed systems. For performance and dependability evaluation extensions of PNs like Generalized Stochastic Petri Nets [1], Stochastic Reward Nets [6, 12] offer not only precise mathematical background but also sophisticated off-the-shelf analysis tools. Although there are also other methodologies for quantitative analysis (like queueing networks [2], stochastic process algebra [3] etc.) Petri nets are still considered to be the most mature in terms of the number and scope of theoretical results, the efficiency of the analysis algorithms and the number of available tools [8]. Accordingly, our choice was the class of Stochastic Reward Nets (SRN). SRNs generalize classical PNs by rewards (various measures) and by assigning guards and distributions of the firing time to transitions.

Our transformation is presented in a modular way, by introducing a set of SRN design patterns. These patterns are assigned to peculiar constructs (like event dispatcher) or concepts (like state hierarchy, synchronization) of the UML statechart formalism, this way they help in decomposing the problem and understanding the proposed solutions. These patterns are combined automatically by using well-defined interfaces and composition rules. The modularity of the definition helps also in proving the properties of the resulting SRN model according to the informal requirements of the UML semantics as defined in the standard [13].

The paper is structured as follows. Section 2 discusses the differences between the semantics of UML statecharts and SRN models, thus analyzing the problems to be solved by the transformation. Section 3 identifies the designs patterns and presents the SRN subnets. Section 4 defines the composition rules of the subnets. The application of the transformation in dependability and performance analysis is discussed in Section 5. The paper is closed by a small illustrative example (Section 6) and a section of conclusion.

2 Semantics of models

In this section we summarize and compare the semantics of the source and target models of our transformation. The discussion of the UML statechart semantics is based on the (informal) UML standard [13] and on the formalization presented in [10].

While analyzing the semantics, we were faced with two problems. The first is, that some aspects of UML semantics are not defined in the standard. In this case we tried to parameterize our transformation by elaborating patterns for different possible cases. The next problem is, that the semantics of UML statecharts with timed state transitions was not formalized yet. While considering the issues of time, we were stuck to the requirements of the untimed case (run-to-completion processing, execution steps).

UML statecharts (SC) were specified by the Object Management Group [13] as a variant of classical Harel statecharts [9]. The semantics of UML statecharts is expressed in terms of a hypothetical machine with the following components:

- An event queue storing events coming from the machine itself or from the environment. The internal structure of the event queue is not specified.
- An event dispatcher selecting one event at a time from the queue. If an event is dispatched, it will be passed to the machine to react to it. When the machine finished its reaction (possible state changes) and reached a stable state, a next event can be dispatched. The selection policy of the dispatcher is not defined.
- A state machine processing the dispatched events. The reaction of the machine is determined by its actual state and the possible transitions triggered by the selected event.

The operation consists of cyclic event dispatching and state changing phases, called steps of the state machine. Steps are characterized by run-to-completion processing of events, i.e. there is no new event dispatched until the previous one is completely processed (the state machine reaches a stable state configuration). During a step, several state transitions can be executed, since the statechart may contain concurrent substates.

Other peculiar aspects of the semantics are discussed in the following sections where the particular transformation patterns are presented.

The source models of the transformation described in this paper are restricted to UML statecharts without history states. Actions are restricted to generation of new events, while events cannot have parameters.

Stochastic Reward Petri Nets (SRN) are a GSPN-like formalism based on an independent semi-Markov reward process [6, 12].
By definition, an SRN is a 10-tuple consisting of:

1. a finite set of places,
2. a finite set of transitions,
3. a finite set of inarcs (from places to transitions),
4. a finite set of outarcs (from transitions to places),
5. an integer weight for every arc,
6. a guard function for every transition,
7. an initial marking,

8. a distribution of the firing time for every transition (it can be exponential, deterministic, Cox etc. or a deterministic value 0 for immediate transition),
9. a priority relation (irreflexive, transitive) among the transitions,
10. a finite set of measures.

The transitions of an SRN will be briefly referred to as SRN transitions, in contrast to the UML transitions.

An SRN transition t is enabled for a given marking if and only if the guard function of the transition evaluates to true, there is no other enabled transition with higher priority, and in the given marking there are not fewer tokens on every place p than the weight of the inarc from the place p to the transition t. When the transition t fires, every place p has in next marking as much token fewer, as the weight of the inarc from p to t, and as much token more, as the weight of the arc from t to p. The weight of a non-existing arc is 0.

Firing of SRN transitions has only local effects, i.e. the firing of a transition depends only on the source places and on the guard and timing of the transition, and modifies only its local environment. There is no central event dispatching, and firings of transitions enabled by the same stimulus cannot be divided into steps. Accordingly, event dispatching, the synchronization of guard evaluation, step completion etc. need extra constructions in the transformation.

The target models of our transformation are SRNs with guarded transitions (immediate or timed). SRNs could be defined including inhibitor arcs, and in/outarcs with marking dependent multiplicity, but our transformation does not necessitate these extensions.

In figures, the guards of transitions will be depicted as expressions in square brackets, placed close to their guarded transitions. A *place name* in a guard, or a "mark(*place name*)" expression is true if and only if the named place is not empty. "!", "&&" and "——" are logical NOT, AND and OR operators, respectively. The guard "[guard]" means an arbitrary guard expression.

3 Design patterns

In this section the following patterns of the transformation are outlined:

Event queue and event dispatcher: The events arriving from the environment or from the state machine itself are collected in the queue and dispatched by the dispatcher one at a time. Event queues provide the interfaces among state machines belonging to different objects. The queue and the dispatcher can be implemented by distinguished objects or by the services of the run-time environment (operating system). The UML standard defines precisely neither the policy of the dispatcher nor the number and distribution of event queues. Accordingly, we will define patterns for several policies and leave it to the designer to specify the details in the UML model (e.g. by using constraints).

Priority of transitions: One important feature of statecharts is the hierarchic structure of states. States can contain substates (only one of them is active at the same time) or concurrent sub-machines (all of them are active if their parent state is active). Transitions of an SC may have their source and target states at different levels of the state hierarchy. Due to the state hierarchy, multiple transitions (triggered by the same event and having source states being active in the current state configuration) may be enabled at the same time. Enabled transitions which have common state(s) to exit (i.e. not in concurrent sub-machines) are in conflict. Some conflicts can be resolved by the priority relation: a transition having source state at lower level has higher priority. From the point of view of the priority, enabled transitions can be represented in the form of a tree according to the state hierarchy. Transitions on different branches of this tree can fire independently, while the conflicts of transitions being on the same path from the root to a leaf are resolved by the priority scheme (the transition being closer to the root has lower priority). Conflicts among transitions emanating from the same state are resolved non-deterministically.

Semantics of timed transitions: The standard UML does not define the semantics of timed transitions, therefore the relationship of guard evaluation and time progress is not specified. We will define various patterns for the possible combinations of timing and guard evaluation.

Synchronization: The transitions of the UML statechart fire in steps, i.e. a stable state configuration is reached only if the maximal set of enabled transitions has already fired. In contrary, SRN reaches a stable state after each firing. Since guards are evaluated in stable states, the behavior of the UML state machine and of the SRN model may differ. Extra constructions are required in the SRN to force consistent evaluation of the guards.

3.1 Event dispatchers

Two patterns for event dispatchers are defined below. One is selecting events from the queue non-deterministically. It is easy to implement with SRNs, and it covers all potential behaviors. Another dispatcher is also elaborated, selecting events in the order of their arrival (FIFO, First In, First Out). It is the intuitive operation of an event queue (but it is not easy to implement it by SRNs). These dispatching policies are adequate for different applications. Both of them can be extended to support multi-level priority dispatching, but this is beyond the scope of this paper.

The patterns include explicit places, where tokens representing incoming events are collected, and other dedicated

places where tokens representing selected events will be placed for further processing.

3.1.1 Nondeterministic event dispatchers

Figure 1 shows the pattern for nondeterministic event dispatchers. Tokens representing incoming events are collected in the places on the left side, thus these places contain the same number of tokens as the number of events of the given kind available in the queue.

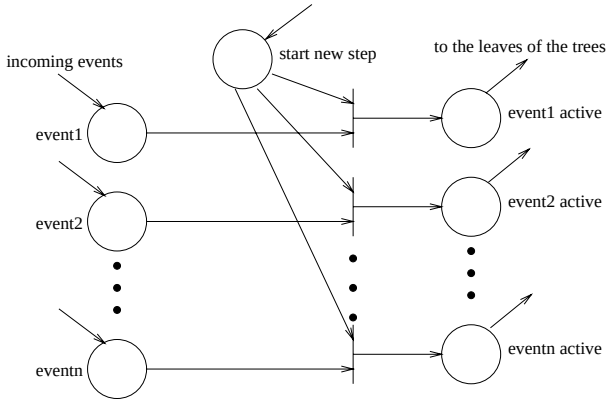

Figure 1. PN pattern of a non-deterministic event dispatcher

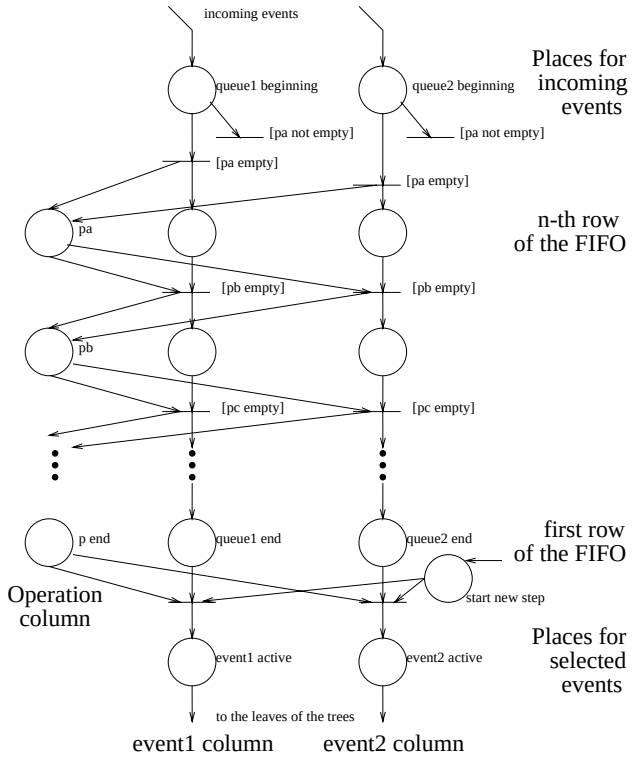

Figure 2. PN pattern of a FIFO event dispatcher

When there is a token at the place *start new step*, a token from a non-empty place on the left side is chosen nondeterministically. It corresponds to the selection of an event by the dispatcher (from the queue). No more events (tokens) can be selected until a new token appears at the place *start new step*, and all non-selected events are preserved. The selected event can be processed by accessing the token on the right side.

This pattern consists of e+l places, e transitions and 3e arcs per per UML states, where e denotes the number of kinds of events handled by the system.

3.1.2 FIFO event dispatchers

Figure 2 shows the pattern for event dispatchers issuing events in the order of their arrival (FIFO). The pattern presented on Figure 2 depicts only two kind of events, but the concept is the same for more events. The input of the queue structure is at the top of the figure, and the output is at the bottom, therefore the tokens will flow downwards in the figure.

There are three columns (of the length of the FIFO) of places: the left-most group is controlling the FIFO structure, the other two (or possible more) groups are for storing the different events. The incoming events arrive at the top of the figure and the just selected one is issued at the bottom. The structure of the pattern guarantees that there are either exactly zero or two tokens in each row. If there are two tokens in a row, one of them is placed in the left-most (i.e. controlling) column. The tokens in this column "fall" to the bottom.

Events arrive to the *beginning* places. If the queue is full, then they will be discarded, else they are placed in the uppermost place of the column corresponding to the type of the event. Simultaneously a token is generated in the uppermost place of the control (left-most) column. The pair of tokens is running downwards to the bottommost row with a free place in the control column. Accordingly, if there is an event on the n-th place of the UML event dispatcher queue, then there is a token in the n-th place (from the bottom) of the operation column and of the column corresponding to the type of the event as well.

Dispatching of events is modeled in the same way as in the case of the non-deterministic event dispatcher. There is an immediate transition for each type of events taking one token from the place *start new step*, one token from the bottom place of the control column and one token from the bottom place (*queue end*) of the corresponding column. One token is put in a place corresponding to the place *event*

i active (Figure 2). The selection of the events is deterministic since there is only a single column (except the control one) with token in the bottom place, and there is always such a column except the queue is empty.

The size of the pattern is as follows (l denotes the length of the FIFO, e denotes the number of kinds of events handled by the system):

Places in this pattern:	$(l+2)e+l$
Transitions:	$(l+2)e$
Arcs:	$4(l+1)e$

3.2 Priority of transitions

One important feature of statecharts is the hierarchical structure of states. A state of an SC can be:

- a basic state, containing no other states,
- an OR-state, containing only substates being active alternatively if the state itself is active,
- an AND-state, containing only concurrent submachines.

Transitions are enabled when their source states are active, their triggering event is dispatched and the guard expressions of the transitions evaluate to true. Two transitions are conflicting when firing of one of them inhibits the other from firing, that is the intersection of the two sets of states they exit is not empty.

Transitions originating from substates of the source state of another transition have higher priority than the other transition. When several transitions are enabled, the maximal non-conflicting set of them (with maximal priority, see below) may fire at the same time in a single step. Each step consists of the following hypothetical phases:

- dispatching an event,
- collecting the enabled transitions,
- selecting a maximal subset of them, where enabled transitions with higher priority must not left out if another transitions with lower priority are therein,
- firing the selected transitions simultaneously.

The priority relation defines a partial ordering relation over the set of the transitions (because there can be source states not containing each other). Partial ordering relations are usually represented as tree structures.

The priority relation of transitions has to be implemented by the transformation. The tree structure offers itself as a good choice: the transitions triggered by the same event can be arranged in a tree corresponding to the hierarchy of the transitions. (Trees are depicted having root at the top and leaves at the bottom, thus the directions “up” and “down” have to be understood accordingly.) A transition with higher priority is located closer to the leaves, and non-conflicting transitions and conflicting ones with equal priorities are located on different arcs of the tree. Compound transitions are mapped to a set of simple transitions.

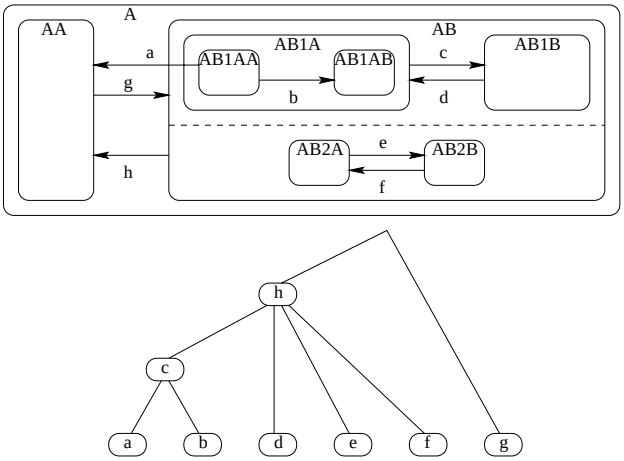

Figure 3. The tree structure of the priority relation

Figure 3 shows a small statechart as an example. 8 transitions (a to g) are presented, all of them being triggered by the same event. (Transitions triggered by other events are not depicted.) The tree structure of the transitions is shown on the bottom.

The structure of the tree strongly depends on the priority structure of the transitions to be transformed, therefore it is not expedient to draw a generic tree structure here. The internal structure of the SRN patterns corresponding to the individual nodes of the tree will be described below.

When an event is selected, the tokens representing the selected event should run through the tree from the leaves to the root. On parallel arcs they run simultaneously, the arcs are synchronized only at the join points. Every transition has to know, whether the transitions with higher priority have “consumed” the event or not, because an enabled transition may only fire if the transitions with higher priority could not fire. In the tree structure, the transitions get the event in the order of their priorities.

Accordingly, the SRN representing the selection of UML transitions is a tree of interconnected sub-SRNs (each of them representing a single UML transition) with an auxiliary control structure. This additional structure consist of two chains of places, where the tokens representing the events can run through the tree. A given token runs on one of the chains, when the event is “not yet consumed” by the transitions on the given arc of the tree, and the token runs on the other chain, when the event is “already consumed”. These chains will be referred to in this paper as chains of

"unconsumed/consumed events".

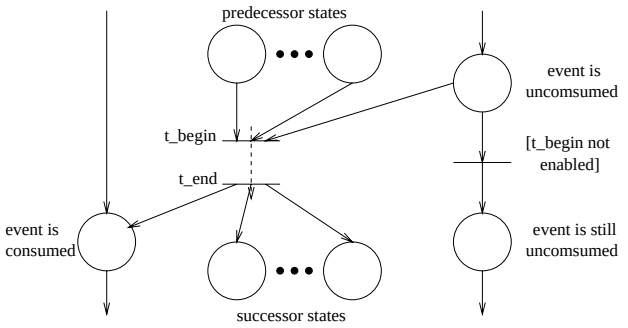

Figure 4. PN pattern of a simple transition

Figure 4 shows the SRN pattern of a simple (i.e. not join) node of the tree. It consists of places representing the following items:

- states to be left when the transition fires (predecessor states),
- states to be entered when the transition fires (successor states),
- the "chain of consumed events",
- the "chain of unconsumed events".

The predecessor states are the source state of the transition and all of its parent states which are not parent states of the target state. They can be identified by the analysis of the structure of the SC. There are also other states to be left, namely the active states of parallel regions of the SC. These states cannot be identified unambiguously by the static analysis of the SC, thus exiting these states necessitates an other construction described later in this paper. These states are not represented in the SRN corresponding to the transition.

Accordingly, firing of an UML transition as represented by the above SRN pattern can result in an inconsistent state. This inconsistency vanishes before completing the step, because

- if there is a transition having (partially) common predecessor (source and parent) states with the given one, then leaving a common predecessor state inhibits the incorrect firing of this transition;
- if there are no common predecessor states, then the result of an incorrect firing vanishes before completion of the step (the results of firing in a parallel region will vanish as the states in this region will be de-activated explicitly at the end of the step).

The successor states of a transition are the states to be entered when the transition fires. This set of states can be unambiguously identified by analyzing the static structure of the SC.

At the beginning of a step, the selected event is "not consumed", i.e. no transition has fired processing that event. Accordingly, the tokens representing the event on the several arcs of the appropriate tree structure of the triggered transitions appear in the "chain of unconsumed events". They will be put on the other side immediately, if a transition fires on the given arc. The dashed arrow in the figure represents some places and transitions corresponding to the timing policy, as will be described in section 3.3.1. Such a sub-SRN corresponding to an UML transition can only fire if the token representing the triggering event appears on the "unconsumed" side.

If there are two conflicting transitions of the statechart enabled at the same time then the firing of the corresponding sub-SRNs happens as follows:

- If one them has higher priority than the other one, then it is placed closer to the leaves of the tree structure, and the sub-SRN corresponding to the other transition can only fire if the event was not consumed by the sub-SRN corresponding to this transition.
- If they have the same priority, then the transitions are placed on different arcs of the tree, and the conflict is resolved by the guards and the firing times of the timed UML transitions.

It can be proved that the properties of the UML SC semantics are satisfied by these patterns, i.e. a sub-SRN corresponding to an UML transition can only fire if the predecessor states of the transition are active, its guard evaluates to true and no transition with higher priority was enabled and triggered.

A joining node of the tree is shown at Figure 5. In these nodes there are no sub-SRNs corresponding to the UML transitions, they only merge the event chains of the subtrees. All of the UML transitions in the subtree have higher priority than any transitions along the common path of the tree above the joining node, therefore "event is unconsumed" applies to this common path if and only if the event was not consumed by any of the transitions of the subtree.

The "event is consumed" applies to the common path when some of the transitions of the subtree have already fired (they had carried over the tokens on the "consumed" chain) and the other transitions could not fire (they passed on the tokens along the chain). This construction ensures that if the token representing the event reaches the root of the tree, no more sub-SRNs corresponding to transitions of the statechart will fire, the *next* and *last* places (see Section 3.3.2) can be synchronized, and the places corresponding to some exited states can be emptied.

In the figure there is an SRN inarc with multiplicity higher than 1. The multiplicity of that arc is equal to the number of the joining arcs in this point. The names "still

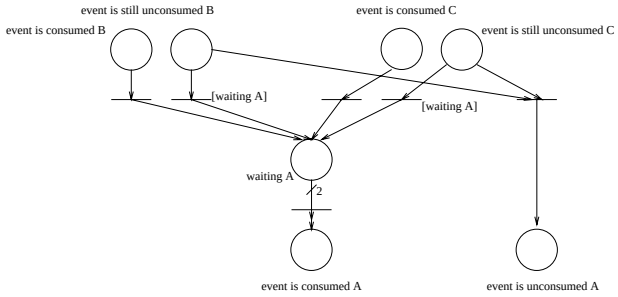

Figure 5. PN pattern of a joining node in the tree structure

unconsumed", "consumed" and "unconsumed" correspond to the names in Figure 4.

The size of the tree pattern is as follows (the duplicated places representing the states of the SC are not counted):

Transitions of the statechart:	t
Joining points of the tree:	j
Joining arcs in a given point:	j_i
Sub-SRNs described in 3.3.1:	t
Additional places:	$3t + j$
Additional transitions:	$t + \sum_{i=1}^{j}(2 + 2j_i)$
Additional arcs outside:	$6t + \sum_{i=1}^{j}(3 + 3j_i)$

3.3 Semantics of transitions

There are differences between the semantics of UML statecharts and of SRNs. First, the relationship of timing and guard evaluation is not specified in standard UML. Second, the execution step of UML semantics requires some synchronization of firings, while in SRN the transitions fire independently.

Below we describe some possible semantics for timed and guarded UML transitions and their transformation patterns. At first, the problems of combining guard evaluation and timing are discussed.

3.3.1 Guards and timing

In our approach, time delay is associated with UML transitions, assuming that this delay is produced e.g. by program code execution or communication delay. Accordingly, the guard expressions have to be evaluated before the firing of the (timed) transitions. Another possible way is to associate the delays to the states, where the evaluation of the guards and the selection of the transitions is preceded by some delay. In our opinion, the former approach fits better to the majority of practical problems.

Figure 6 shows three different (alternative) implementations of the combination of timing and guard evaluation. They may fit to different applications. Since only enabled UML transitions can be selected for firing, the first transitions of each pattern below must be guarded. This guard contains the guard of the appropriate UML transition extended by a conjunctive term to express that the transition can only fire if the appropriate state was active before the actual step.

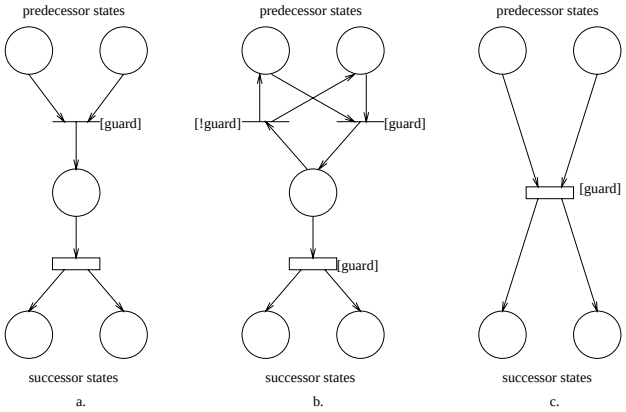

Figure 6. Models for combining guards and timing

The three alternatives are as follows:

- The selection of the transitions is irrespective of timing. (a.)
- The guard has to be true during the delay else the transition will be deselected. (b.)
- The "fastest" enabled transition wins. (c.)

The three figures show sub-SRNs corresponding to the transitions of the statechart, as introduced in the previous section. The predecessor and successor states are shown, but the event chains are not.

The types and parameters of the timed SRN transitions correspond to the types and parameters of the corresponding SC transitions. No other timed transitions are in the patterns, since according to the UML semantics, the event selection, guard evaluation and the selection of the fireable subset of the enabled transitions happen immediately, i.e. in zero time. The timing policy (resampling, race with age/enabling memory, ...) is determined by the designer (and must be implemented by the SRN-tool used for the analysis).

Event sending by the transitions is implemented by outarcs from the timed SRN transitions to the appropriate places of the event dispatcher.

The number of model elements in the pattern is as follows (the duplicated places representing the states of the statechart are not counted to the size of the patterns, t_p

denotes the number of predecessor states, t_s denotes the number of successor states of the given transition):

Model	Places	Transitions	Arcs
a	1	2	$t_p + t_s + 2$
b	1	3	$2t_p + t_s + 3$
c	0	1	$t_p + t_s$

3.3.2 Synchronization

The UML semantics requires the evaluation of the guards of the transitions at the beginning of a step, before firing of any transition. There is no doubt that the guards refer to the consistent state configuration before the actual step. In SRNs, the guard of a transition will be evaluated just before the given transition fires, the evaluation is not scheduled to the beginning of a "step" and the results are not stored. In SRNs it is possible, that some transitions have already fired before the guard expressions of another transitions are evaluated. To the correct evaluation of guards the last stable state configuration of the state machine (i.e. the state before the actual step) must be recorded. To do that, the places representing the states of the SC are duplicated. For a state A there is a place A containing a token if and only if the state A was active just before the actual step (called in the following *last place*), and there is an other place A' containing a token if and only if the state A will be active after the actual step (called *next place* in the following).

The places "predecessor states" and "successor states" in Figure 4 depict the *next* places, while the guards of the appropriate transitions in the sub-SRNs corresponding to the UML transitions are expressions over marking of the places recording the last stable state of the system (i.e. *last* places). The contention is for the tokens of the *next* places, while the *last* places provide a consistent guard evaluation during the firing of the guarded transitions.

This concept necessitates a synchronization of the duplicated places at the end of each step. The SRN pattern of the synchronization is shown in Figure 7.

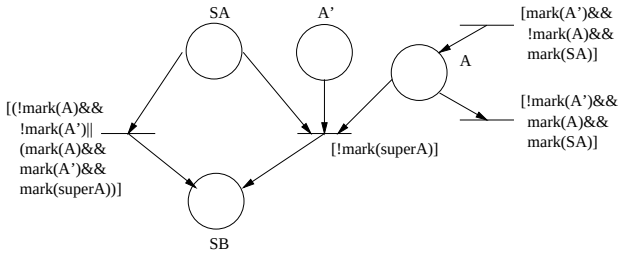

Figure 7. Synchronization of the duplicated places

The pattern shows the synchronization of the places representing the state A of a statechart. There must be such a pattern for each state.

The places A and A' represent the given state. There is a token in A if and only if the state A of the SC was active just before the actual step, and there is a token in place A' if and only if the state A of the SC will be active after the actual step. The places SA and SB are places of a synchronization chain described below. The place name *superA* should be replaced by the place name according to the SC state directly containing the state A.

This pattern not only synchronizes the duplicated places, but also corrects transient inconsistencies in the markings. Due to the incompleteness of identifying the dynamically changing set of active states when an SC transition fires, the tokens must be removed from places representing states considered to be inconsistently active, since their parent states are inactive (see an example below).

Remember that the predecessor states on Figure 4 are only the source and parent states of the SC transition, which are to be exited. However, there are other states also to be exited, namely the active substates, and the active states of parallel regions of states to be exited. Since they cannot be identified statically, these states were not emptied when the predecessor states were exited. This inconsistency must be resolved at the end of the step. Note that this vanishing problem does not affect the result of the step.

Example: On figure 3 a small statechart is presented. The predecessor states of the transition *a* are *AB1AA*, *AB1A* and *AB*. If *a* is enabled then either *AB2A* or *AB2B* must be active (since their parent state *AB* is active). It cannot be identified statically, which of them is active at the given situation, therefore they do not appear in the set of predecessor states of *a*. Before the end of the step when *a* fires, the active one of them must be exited, because their parent state *AB* was exited.

When the token representing the selected event reaches the root of the tree of the triggered transitions, it is passed to a synchronization chain. This chain controls the synchronization of the duplicated places. All states of the SC are included in this chain, where every state precedes all of its substates, otherwise the order is arbitrary. In the SRN model, the synchronization chain is the chain of places corresponding to the SC states.

The synchronization of the duplicated places could happen independently, but this non-deterministic order would produce a large state space of the SRN without any further advantages. The fixed ordering avoids this kind of state space explosion.

Figure 7 shows the synchronization pattern of state A. Until there is no token in the place SA, nothing can happen. When there is a token in this place, it cannot be removed until:

- There is a token in both places A and A' but there is no token in the place representing the direct parent state of state A (the place pair corresponding to that state is already synchronized). In this case both A and A' will be emptied, and the token from the place SA will be passed to the place SB (for synchronizing the next state in the order of the synchronization chain).
- The markings of the places A and A' are the same, with respect to that both of them can have a token if and only if there is a token in the place representing the direct parent state of A.

Note that the places corresponding to SC states never have more than one token in these SRN patterns.

The synchronization of the places A and A' can happen by the help of the two transitions on the right side of the figure, if there is a token in SA.

In the synchronization subnet belonging to the last state of the chain, SB is replaced by *start new step*.

This pattern consists of 1 place, 4 transitions and 8 arcs per per UML states.

4 Composition of subnets

By analyzing the structure of a given statechart the above mentioned subnets are constructed based on the design patterns. The subnets are connected with each other according to the interface places identified by the same name in the patterns.

The necessary number of patterns is the following:

- The number of event queues and the type of the event dispatcher(s) is defined by the designer (additional information is attached to the UML model).
- There are as many transition hierarchy trees as the number of events handled by the transitions of the statechart.
- The number of sub-SRNs representing transitions is the same as the number of transitions in the model.
- Each state of the statechart is represented by a pair of places in the SRN.
- For each state of the statechart, there is a synchronization subnet.

The initial state of the SRN is defined as follows. If the event queue contains events in the initial state then these events are represented by the initial marking of the appropriate places. The initial state configuration of the SC has to be mapped to the SRN by inserting tokens into the corresponding place-pairs. The initial marking of the place *start new step* has to be one.

The external environment can be modeled (in closed systems) by separate UML statechart(s) which will be transformed to SRNs with outarc(s) to the appropriate places of the event queue(s).

5 Model analysis

The model can be analyzed by standard SRN tools. In certain cases analytic solution is possible, otherwise simulation has to be performed. If a steady state exists then steady state measures can be computed, otherwise transient analysis can be executed.

The results of the analysis of the SRN (and so of the transformed UML model) are, for example,

- the reachable state and state configurations of the system,
- the expected probability that a state is active,
- the expected value of the throughput of a transition,
- the expected probability that a transition is enabled,
- the expected probability that a transition fires.

These results can be utilized to gain both performance and dependability measures of the model.

Simple performance measures (throughput, utilization) can be derived directly from the above presented results. In more complex cases, user-defined reward functions can also be used (expressed in the UML model in the form of structured comments).

Dependability-based analysis in this framework requires explicit modeling of faulty behavior. The approach presented in [7] can be followed. The error model based on state perturbations (unintended state transitions, loss of messages) can be applied. Erroneous states can be included explicitly in the model. The duplication of places representing SC states (Section 3.3.2) can be directly utilized for modeling of failures (loss of synchrony). The possibility of event processing enables to model error propagation, communication errors, external erroneous messages reaching the system etc. The corresponding states and events can be distinguished in the UML model by stereotypes, the necessary probabilistic information can be included in the form of tagged values (corresponding to transition firing times in the SRN model). The enriched SC can be transformed to an SRN as described in the previous sections. The resulting SRN can be analyzed in special consideration of the distinguished (erroneous) states.

The analysis of the probability of erroneous states leads to reliability (if no repair is modeled) and availability figures (if repair is modeled). Analogously, safety figures can be derived by distinguishing the unsafe states in the model. Performance and dependability characteristics can be combined e.g. by comparing the performance in fault-free and erroneous cases.

6 Examples

To give a formal proof of the equivalence of the arising SRN and the original SC is beyond the scope of this paper.

In this section only an illustrative example is presented: a small system, several statechart models of it and the SRNs representing them.

The example is a variation of a production cell model [11]. The system contains a press that processes metal blanks, a robot with two arms for loading and unloading the press, and a rotary table for positioning the blanks for the robot. A UML-model of an extended version of this example is given in [5].

A high-level model of the production cell consists of 6 statecharts. Each of them has its own event queue and event dispatcher with non-deterministic dispatching policy. The statecharts have 4,2,2,2,2,2 states, 9,2,2,2,2,2 transitions and 5,2,2,2,2,2 events, respectively. The SRN representing this statechart model consists of 103 places, 112 transitions (82 guarded, 20 timed), 142 inarcs and 144 outarcs.

A more detailed model consists of one single statechart with 15 concurrent states containing 50 substates, and 68 transitions triggered by 42 events (14 timer events). A single global event queue is supposed with non-deterministic dispatching policy. This statechart was transformed to an SRN with 373 places, 472 transitions (304 guarded, 82 timed), 547 inarcs and 558 outarcs.

7 Conclusion

We presented a method which allows quantitative performance and dependability analysis of systems modeled by using UML statechart diagrams. Our transformation from statecharts to Stochastic Reward Nets covered a large subset of model elements including event processing, state hierarchy and transition priorities. By using the transformation and analyzing the resulted SRN performance and dependability measures can be computed. Since the analysis is based on a detailed model of the system, in the case of complex systems this kind of analysis should be restricted to core critical parts of the system.

The transformation was presented in the form of design patterns. The properties of the resulting SRN satisfy the requirements defined in the UML standard. The number of places and transitions in the generated model is proportional to the number of model elements in the statechart. The generated number of states (state space of the underlying Markov chain) corresponds to the number of state configurations of the UML model.

8 Acknowledgements

The authors thank Prof. M. Dal Cin (IMMD3, FAU Erlangen-Nuremberg, Germany) for supporting this work and enhosting the first author. The research has benefited from helpful comments from A. Pataricza (TU-Budapest).

References

[1] M. Ajmone Marsan. Stochastic Petri nets: An elementary introduction. In G. Rozenberg, editor, *Advances in Petri Nets*, LNCS 424, pages 1–29. Springer Verlag, 1991.

[2] F. Bause, P. Buchholz, and P. Kemper. Hierarchically combined queueing Petri nets. In *Proc. 11th Int. Conf. on Analysis and Optimization of Systems, Discrete Event Systems*, Sophie-Antipolis, France, June 1994.

[3] M. Bernardo and R. Gorrieri. Extended Markovian process algebra. In U. Montanari and V. Sassone, editors, *CONCUR'96, 7th Int. Conf. on Concurrency Theory*, LNCS 1119, pages 315–330, Pisa, Italy, 1996. Springer Verlag.

[4] A. Bondavalli, M. Dal Cin, D. Latella, and A. Pataricza. High-level Integrated Design Environment for Dependability (HIDE). In *Proc. Fifth Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-99)*, Monterey, California, USA, November 18-20. 1999.

[5] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analysis for supporting design decisions in UML. In *Proc. HASE'99, Fourth IEEE Int. Symposium on High Assurance Systems Engineering*, Washington DC Metropolitan Area, USA, November 17-19. 1999.

[6] G. Ciardo, A. Blakemore, P. Chimento, J. Muppala, and K. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. In *Linear Algebra, Markov Chains and Queueing Models*. Springer Verlag, 1992.

[7] M. Dal Cin, G. Huszerl, and K. Kosmidis. Quantitative evaluation of dependability critical systems based on guarded statechart models. In *Proc. HASE'99, Fourth IEEE Int. Symposium on High Assurance Systems Engineering*, Washington DC Metropolitan Area, USA, November 17-19. 1999.

[8] S. Donatelli, J. Hillston, and M. Ribaudo. A comparison of performance evaluation process algebra and generalized stochastic Petri nets. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, Duke University, North Carolina, USA, October 3-6. 1995.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[10] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1, 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Syst.*, pages 331–347, Firenze, Italy, February 1999.

[11] C. Lewerentz and Th. Lindner, editors. *Formal Development of Reactive Systems*, volume 891. of *Lecture Notes in Computer Science*. Springer, 1994.

[12] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Stochastic reward nets for reliability prediction. *Commun. in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.

[13] OMG. *UML Semantics, version 1.1*. Object Management Group, September 1997.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.