Quantitative Analysis of UML Statechart Models of Dependable Systems

GÁBOR HUSZERL[1], ISTVÁN MAJZIK[1], ANDRÁS PATARICZA[1],
KONSTANTINOS KOSMIDIS[2] AND MARIO DAL CIN[2]

[1]*Department of Measurement and Information Systems,*
Budapest University of Technology and Economics, Hungary
[2]*Department of Computer Science III (Computer Structures),*
Friedrich-Alexander University of Erlangen-Nuremberg, Germany
Email: huszerl@mit.bme.hu

The paper introduces a method which allows quantitative dependability analysis of systems modeled by using the Unified Modeling Language (UML) statechart diagrams. The analysis is performed by transforming the UML model to stochastic reward nets (SRNs). 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 patterns. Performance-related measures can be directly derived using SRN tools, while dependability analysis requires explicit modeling of erroneous states and faulty behavior.

Received 22 November 2000; revised 26 November 2001
Handling Editor: I.-R. Chen

1. INTRODUCTION

Formal modeling and analysis techniques of modern computer controlled systems are becoming more and more important. Well-specified and easy-to-use design languages and environments are required that enable multi-aspect analysis and verification of the designs. In critical systems like transportation or production systems not only does the functional correctness have to be analyzed, but also the reliability, availability, safety and performability. The analysis is especially important in the early design phases, since modifications and re-design are extremely costly if an inadequacy is detected in the later phases of the development.

A core requirement for dependability-critical systems is the ability to cope with faults. It is important that this non-functional property can be validated before the system is licensed for use in applications that affect, for instance, human life. This requires a quantitative analysis, which deals with reliability, availability and safety. For such an analysis, in addition to the system's function, the modeling of faults and their treatment is also necessary, including the effects of both possible internal faults and faults concerning the system's interaction with its environment (via sensors and actuators). In this way the effects of a component fault on system behavior, the error coverage and the recovery cycle can be analyzed.

Nowadays a wide variety of formalisms, languages and analysis techniques are offered to the designer. From the viewpoint of design re-use and tool support, standardized design languages are preferred. The Unified Modeling Language (UML) [1] provides a visual notation (standardized by the Object Management Group [2]) for expressing the artifacts of complex distributed systems ranging from embedded systems to business applications. UML is supported by a wide variety of well-established tools and environments, offering services for specification, design refinement and automatic code generation. In recent years, several methods have been elaborated to enable us to also carry out formal analysis of UML-based designs. Among others, problems of system-level dependability modeling, formal verification and performance analysis of (subsets of) UML models have been addressed in [3].

Our work is focused on the quantitative dependability analysis of the UML behavioral models of embedded systems. The dynamic behavior of the system is given in UML by statechart diagrams [2], an object-oriented mutation of classical Harel statecharts [4]. 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 dependability analysis to be carried out, if the model is extended with explicit categorization of failure states/events and probabilistic information. Although the UML notation was not designed for these purposes, its standard mechanisms enable one to extend the model with both timing/stochastic information (in the form of tagged values) and classification of model elements (in the form of stereotyped states and events).

Evaluation of embedded systems tends to be very complex. Therefore, when modeling embedded systems a trade-off has to be made between the degree of detail in modeling and the degree of possible automation of the analysis. This leads us to define a subclass of UML statecharts comprising so-called guarded statecharts (GSCs) [5]. GSC models are well suited for modeling of embedded systems where synchronization among components can be described solely by Boolean predicates on the active states of concurrent components [6]. This kind of modeling can be considered as a higher-level, behavioral view of the system. However, GSC models do not support more implementation-related details like event processing, a concept which may be of crucial importance in modeling the real architecture of a system. Moreover, this formalism prohibits the use of state hierarchy, one of the most useful concepts in statechart diagrams. Accordingly, we also cover event processing and state hierarchy. Reaching the level of almost full UML statecharts in this way, the modeler is allowed to prepare more compact and intuitive models; however, the complexity, time and resource requirements of the analysis increase.

The analysis is based on a transformation from these statechart models to Petri nets (PNs) with timing and stochastic extensions. PNs are a widely accepted formalism for modeling and analysis of distributed systems. For performance and dependability evaluation, extensions of PNs with firing time distributions of transitions, like generalized stochastic Petri nets (GSPNs) [7] and stochastic reward nets (SRNs) [8, 9], offer not only a precise mathematical background but also sophisticated analysis tools. Although there are also other methods for quantitative analysis (like queuing networks [10], stochastic process algebra [11], etc.), Petri nets are still considered to be the most mature in terms of the scope of the theoretical results, the efficiency of the analysis algorithms and the number of available tools [12]. SRNs generalize classical PNs by rewards (various measures) and by assigning guards and distributions of the firing time to transitions. Accordingly, we choose SRNs for our analysis.

The paper is structured as follows. The next section introduces the approach of model transformation. In Section 3 the GSC models and the corresponding model transformation are presented. In Section 4 we extend the model with event processing and state hierarchy and identify the corresponding model transformation patterns. Modeling of faults, as a crucial step in dependability analysis, is discussed in Section 5. The application of the transformations in dependability analysis is discussed in Section 6. An illustrative example is presented in Section 7. Section 8 contains the conclusions.

2. ANALYSIS FORMALISMS AND TOOLS

UML models are not directly amenable to quantitative dependability analysis. Therefore, a method has to be introduced which generates a mathematical model that can be evaluated by existing tools. The results of the analysis can then be back-annotated to the UML model. In this way the problems of both the manual re-modeling of the system and the need for expertise in formal mathematics are eliminated. This approach originated in the HIDE (High-Level Integrated Design Environment for Dependability[1]) project that proposed a general environment to integrate the various model transformations from UML models towards specific mathematical formalisms and analysis tools [3].

In our case the quantitative analysis of UML statechart diagrams is performed by transforming them to SRNs. The HIDE environment is utilized to define and implement the transformation.

SRNs are a GSPN-like formalism based on a semi-Markov reward process [8, 9]. By definition, an SRN is a 10-tuple consisting of:

1. a finite set of places;
2. a finite set of transitions (the transitions of an SRN will be briefly referred to as SRN transitions, in contrast to the UML 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 zero for immediate transition);
9. a priority relation (irreflexive, transitive) among the transitions;
10. a finite set of measures.

An SRN transition t is enabled for a given marking if and only if (i) the guard function of the transition evaluates to true, (ii) there is no other enabled transition with higher priority and (iii) on every place p there are not fewer tokens in the given marking than the weight of the inarc from the place p to the transition t. When the transition t fires, the number of tokens on every place p is increased by the weight of the inarc from p to t, and decreased by the weight of the outarc from t to p. The weight of a non-existing arc is zero.

The target models of our transformation are SRNs with guarded transitions (immediate or timed). SRNs could be defined including inhibitor arcs, but our transformation does not necessitate this extension.

Two SRN tools, SPNP [13] and PANDA [14], were used in our analysis environment (both of them have a compatible input format called CSPL, the C-based SPN Language). PANDA allows one to annotate transitions with guards and to use state-dependent capacities for arcs. Moreover, PANDA accepts not only exponential distribution functions, but also non-exponential ones (Erlang-k, gamma, Weibull, normal, lognormal, hyperexponential, etc.). Dependability measures can be specified by reward functions. To this end, a reward concept is available based on reward rates

[1]The HIDE framework was developed under EU contract ESPRIT Open LTR 27439, participants were the University of Erlangen-Nuremberg, Consortio Pisa Ricerche—Pisa Dependable Computing Centre, Technical University of Budapest, MID GmbH and INTECS Sistemi S.p.A.

and impulse rewards combining knowledge of the net model and the state space. (The net view is not lost when defining reward functions on the state space.) Reward functions are built from so-called characterizing functions like *mark(place)* which deliver the number of tokens in an SRN place. PANDA computes the expectation value of a reward function (e.g. availability or throughput) as well as accumulated rewards.

3. GUARDED STATECHART MODELS

GSCs are a sub-class of UML statecharts. GSCs represent finite-state machines and describe the behavior of objects in response to external stimuli (such as sensor signals), modeling state-driven system behavior. The main elements of a GSC are states (container states, basic states and initial states) and transitions with guards. Labels of transitions describe timing information, e.g. the arrival distribution of signals, or static information, e.g. the probabilities of possible outcomes. These labels can be provided as UML tagged values in the form, e.g., 'rate = 10' or 'weight = 0.6'.

3.1. The GSC formalism

Given a set E of external event variables, a GSC is a finite set A of state transitions and a finite set S of states. Transitions include the following elements:

- the *trigger* is a Boolean expression of atomic predicates over event variables;
- the *guard* is a Boolean expression of predicates *in(state)* where *in(state)* evaluates to true if *state* is the actual state of the GSC or of some concurrent GSC;
- the *set of target states* to be entered.

When state transitions are depicted graphically, they are labeled with labels of the form `tr[guard]`, where `guard` is (the name of) a guard and `tr` is (the name of) a trigger.

GSCs are not hierarchic—rather, there are only two levels. At the upper level there are container states that describe concurrent behavior by comprising simple state machines.

With GSCs non-deterministic behavior can also be modeled. This is important, since although the software of embedded systems is completely deterministic, the system can not know if and when external events or faults will occur.

The execution of the transition is atomic and instantaneous, if its trigger and its guard evaluate to TRUE. The execution effects the non-deterministic choice of exactly one state of the set of target states as the next state of the GSC. A guard expression of a state machine M may not contain predicates of states of M; it may, however, contain state predicates of a concurrent machine of the GSC.

An example of a transition is

```
startsignal_on [in(L.up) && in(N.ready)]
```

and the target states are

```
{M.ready, M.waiting}
```

Here `M.up`, `M.ready` and `M.waiting` are states of GSC M, `L.up` is a state of the concurrent GSC L and `N.ready` is a state of the concurrent GSC N; `&&` is the logical AND operator.

Guards can be considered as high-level abstractions of synchronization mechanisms. Outputs are considered to be part of the state in which they occur.

3.2. Modeling with GSCs

In this section we indicate how GSCs can be used to model the behavior, e.g., of an embedded control system.

Using GSCs we can abstract continuous signals to discrete signals by assuming a finite set of critical attribute values. For example, it is only important to observe whether a robot arm is directed in a position allowing for unloading or pointing towards a press; all intermediate positions can be collapsed into a single third value. This way, we model sensor and actuator signals via states. A state representing an actuator signal being active means that the actuator is set to a certain discrete value. Analogously, if a component is in a state which represents a sensor signal, it means that this sensor is set. In GSC models, hardware and software components are only allowed to communicate via such sensor and actuator states. This interaction is expressed by guard expressions containing predicates over sensor or actuator states (so-called public states). Similarly, interactions between tasks of the control software are also modeled by guarded state transitions. This corresponds to an asynchronous synchronization pattern between tasks. This pattern is inherently multi-threaded, because it models a message being passed to another object without the yielding of control [15].

The following steps lead to a GSC model of an embedded system and its environment, which comprises controllers and the controlled units interacting by sensors and actuators.

1. Prepare the component models. Specific states (the public states) describe the events the system components (controllers and controlled units) generate or respond to. These states represent, e.g., sensor and actuator signals. The controllers manage disjoint sets of actuator signals. The model of controlled units, usually, need not be very detailed, since its only purpose is to restrict the state space of the controllers to reasonable state transitions and to inform the controllers about sensor or actuator failures.
2. Specify guards for state transitions. These guards represent the component's inferred knowledge about its environment, i.e. about the actual public states of certain system components, and determine the response of the components to this knowledge.
3. Specify the rates or weights of state transitions. Rates label timed transitions and specify the mean transition time. Weights label immediate, timeless transitions and specify the branching probability in case of conflicts.
4. Specify the dependability measures. These measures can be expressed in terms of reward functions [8],

assigned to the UML model in the form of structured comments.

3.3. From GSCs to SRNs

For dependability analysis the GSC models are transformed to SRN models amenable to mathematical analysis. The transformation neglects the concurrent container states, since they have no counterparts in the SRN structure. The following three simple patterns are used.

1. The basic states are represented as SRN places. The places hold the names of the basic states. The initial marking of a place is 1 if there is an initial transition in the GSC leading to the corresponding state. Otherwise the initial marking is 0.
2. State transitions labeled with rates are transformed to timed SRN transitions with the same rates. Guards and triggers become guards of SRN transitions.
3. State transitions labeled with weights are transformed to immediate SRN transitions with the same weight. Immediate transitions have priority over timed transitions. The weights of conflicting immediate transitions are normalized so that they become branching probabilities.

Additional SRN transitions are generated for loss of signals or generation of spurious signals (see Section 5). The modeler has only to specify the rates.

This way we obtain a set of topologically isolated subnets which interact by guards. This approach requires fewer modeling elements than a single SRN without guards and, thus, makes the model more comprehensible.

4. EVENT PROCESSING AND STATE HIERARCHY

Extending the GSC model with event processing and state hierarchy needs a thorough analysis of the semantics of UML statecharts. In this section first we summarize and compare the semantics of the source and target models of the transformation. The discussion of the UML statechart semantics is based on the (informal) UML standard [2] and on the formalization presented in [16].

In the next subsection the transformation from UML statecharts to SRNs is discussed. Our transformation is presented in a modular way, by introducing a set of SRN *transformation patterns*. These patterns are assigned to elements, peculiar constructs (like event dispatcher) and concepts (like state hierarchy, synchronization) of the UML statechart formalism. This approach helps to decompose the problem and understand the proposed solutions. These patterns are combined automatically by using 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 [2].

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

4.1. Semantics of models

While checking the semantics, we were faced with two problems. The first was 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 was that the semantics of UML statecharts with timed state transitions have not yet been formalized. While considering the issues of time, we were stuck with the requirements of the untimed case: run-to-completion processing and execution steps.

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 in UML.
- 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 has 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 in UML.
- A state machine processing the dispatched events. The reaction of the machine is determined by its actual state configuration and the possible transitions triggered by the selected event.

The dynamic operation consists of cyclic event dispatching and state changing phases, which are 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 sub-machines. 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 be omitted if other transitions with lower priority are included;
- firing the selected transitions (the order is not specified).

Several other peculiar aspects of the semantics are discussed in the following subsections where the particular transformation patterns are presented.

Event queues and event dispatchers. 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 does not define precisely the policy of the dispatcher or 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 stereotypes).

Hierarchy of states and 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 a statechart 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 caused by 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.

Step semantics. 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 contrast, an SRN reaches a stable state after each firing. Since guards are evaluated in stable states, the behavior of the UML state machine and that of the SRN model may differ. The consistent evaluation of the guards has to be forced in the SRN.

The main distinguishing feature of the semantics of UML statecharts and those of the SRN is that the 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 thus 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 and the step completion need extra constructions in the transformation.

```
procedure build_nondet_dispatcher(event e) {
   add_place(<name>=e+"0");
   add_place(<name>=e+"1");
   add_trans(<name>="disp_"+e, <type>=immediate);
   add_iarc(<tr_name>="disp_"+e, <pl_name>="READY");
   add_iarc(<tr_name>="disp_"+e, <pl_name>=e+"0");
   add_oarc(<tr_name>="disp_"+e, <pl_name>=e+"1");
   add_trans(<name>="split_"+e, <type>=immediate);
   add_iarc(<tr_name>="split_"+e, <pl_name>=e+"1");
}
```

ALGORITHM 1. Subnet for non-deterministic event dispatching.

4.2. Transformation patterns

The general transformation patterns introduced below were presented first in [17]. In this section we formalize these patterns and demonstrate their typical application to subnets corresponding to the example in Section 7. The patterns are described using a pseudo-code.

The composition rules of the patterns and the generation of the initial marking of the composed SRN are described in Section 4.7. They together build the full algorithm of the transformation. Based on this, the transformation tool identifies the UML model elements and applies the patterns automatically, constructing in this way an SRN corresponding to the UML statechart model.

For the sake of simplicity, it is supposed that each element of the UML model has its own unique identifier and this identifier is reused when giving names to the elements of the SRN model. This method also facilitates reading by visualizing the correspondence. In the figures, the guards of transitions will be depicted as expressions in square brackets, placed close to their guarded transitions. A *place name* in an SRN guard or a *mark(place name)* expression is true if and only if the named place contains tokens. '!', '&&' and '| |' are logical NOT, AND and OR operators, respectively. The guard `[guard]` means an arbitrary guard expression. The function names used in the algorithms are to be interpreted as follows: *parentstate(s)* is a state having an element *s* in its *subvertex* set. *superstate of s* is true for a state if it is *parentstate(s)* or it is *parentstate(parentstate(s))*, and so on recursively until the top state of the statechart is reached. *isRegion(s)* is true for a state if its derived Boolean value *isRegion* (part of the UML metamodel) is true (i.e. it is a direct substate of a concurrent superstate).

4.3. Event queue and dispatcher

We have defined two patterns for event dispatchers [17]. One is selecting events from the queue non-deterministically. It is easy to implement with SRNs and it covers all potential behavior. Another dispatcher is also elaborated, selecting events in the order of their arrival (FIFO, first in, first out). These dispatching policies are adequate for different applications. They can both be extended to also support multi-level priority dispatching.

The generation of the SRN pattern of a non-deterministic event dispatcher is shown by Algorithm 1.

Figure 1 depicts an example subnet corresponding to this pattern. Tokens representing the events of the transformed UML model (*ask*, *down* and *up*) are collected in places *ask0*,

```
procedure build_fifo_dispatcher(event e) {
   add_place(<name>=e+"0"); add_place(<name>=e+"1");
   add_trans(<name>="disp_"+e, <type>=immediate);
   add_trans(<name>="discard_"+e, <type>=immediate, <guard>='"queue_"+n is not empty');
   add_iarc(<tr_name>="discard_"+e, <pl_name>=e+"0");
   add_iarc(<tr_name>="disp_"+e+n, <pl_name>=e+"0");
   for (i=0 to n) {
      add_place(<name>=e"_queue_"+i);
      add_trans(<name>="disp_"+e+"_"+i, <type>=immediate, <guard>="'queue_"+i is empty');
      add_oarc(<tr_name>="disp_"+e+"_"+i, <pl_name>="queue_"+i);
      add_oarc(<tr_name>="disp_"+e+"_"+i, <pl_name>=e+"_queue_"+i);
      if (i>0) {
         add_iarc(<tr_name>="disp_"+e+"_"+(i-1), <pl_name>="queue_"+i);
         add_iarc(<tr_name>="disp_"+e+"_"+(i-1), <pl_name>=e+"queue_"+i);}}
   add_iarc(<tr_name>="disp_"+e, <pl_name>="queue_0");
   add_iarc(<tr_name>="disp_"+e, <pl_name>=e+"queue_0");
   add_iarc(<tr_name>="disp_"+e, <pl_name>="READY");
   add_oarc(<tr_name>="disp_"+e, <pl_name>=e+"1");
   add_trans(<name>=split_"+e, <type>=immediate);
   add_iarc(<tr_name>="split_"+e, <pl_name>=e+"1");
}
```

ALGORITHM 2. Subnet for FIFO event dispatching.

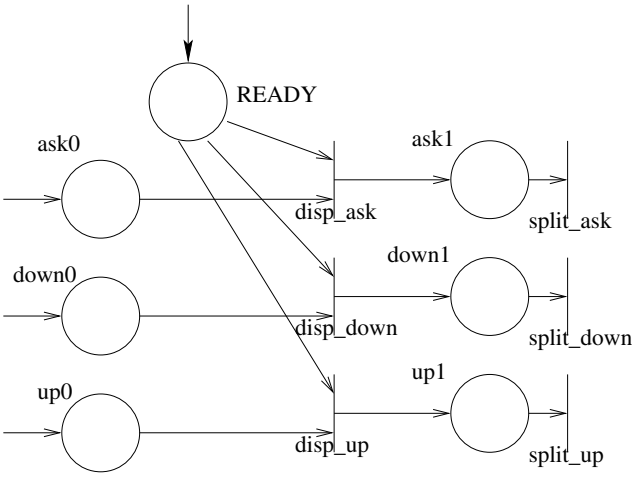

FIGURE 1. SRN pattern of a non-deterministic event dispatcher.

down0 and *up0*, respectively (these events are generated by actions). At the end of a step, a token appears at the place *READY* and a token from a non-empty place on the left-hand side is moved to a place representing the selected event (*ask1*, *down1* or *up1*). It corresponds to a non-deterministic selection of an event by the dispatcher. All non-selected events are preserved and no more events (tokens) can be selected until a new token appears in the place *READY*. The selected event can be processed by accessing the token on the right-hand side. The transitions *split_ask*, *split_up*, etc. are necessary to generate tokens to be processed when the event triggers concurrent UML transitions.

The transformation pattern for the FIFO event dispatcher is more complex: Algorithm 2 contains its formal description and Figure 2 shows a subnet belonging to the pattern.

The subnet depicts only two events of the transformed UML model (*up* and *down*), 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. Here the length of the queue is three.

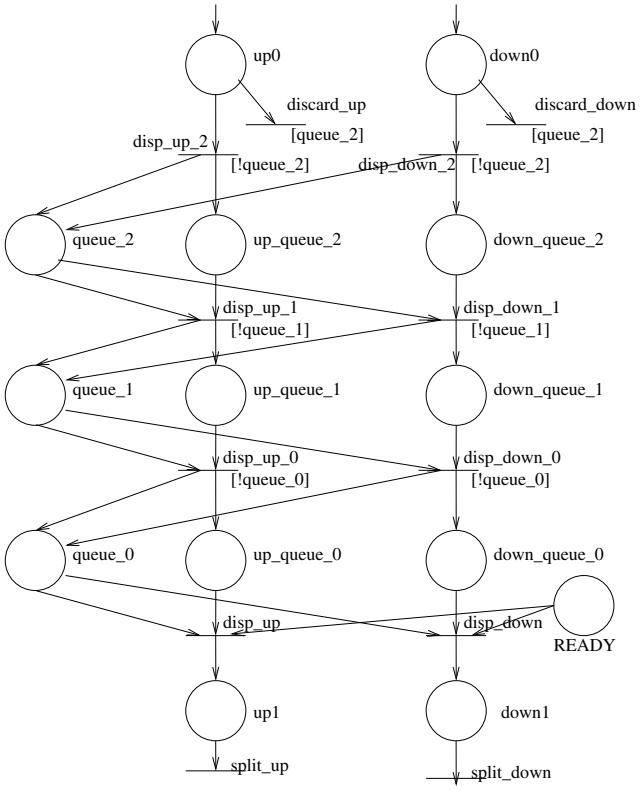

FIGURE 2. SRN pattern of a FIFO event dispatcher.

There are three columns (of the length of the FIFO) of places: the left-most group controls the FIFO structure; the other two groups store the different events. The tokens representing the incoming events arrive at the top of the figure at places *up0* and *down0*, and the one just selected is issued at the bottom in place *up1* or *down1*. If the queue is full, the incoming tokens will be discarded (by transitions *discard_up* and *discard_down*), otherwise they are inserted into the upper-most places of the control (left-most) column and of the column corresponding to the type of the event. The pair of tokens runs downwards to the bottom-most row with a free place in the control column. Dispatching of

```
procedure generate_tree(event e) {
   new_relation(<name>=T, <type>=unidirectional, <fields>=(ancestor, descendant));
   for (each transition t1 of the SC triggered by e) {
      for (each transition t2 of the SC triggered by e) {
         if (source(t1) is superstate of source(t2)) {
            add_relation(<relation>=e <ancestor>=t1, <descendant>=t2);}}}
   for (each transition t1 of the SC triggered by e) {
      for (each transition t2 of the SC triggered by e) {
         for (each transition t3 of the SC triggered by e) {
            if (in_relation(<relation>=e <ancestor>=t1, <descendant>=t2) AND
                in_relation(<relation>=e <ancestor>=t1, <descendant>=t3) AND
                in_relation(<relation>=e <ancestor>=t3, <descendant>=t2)) {
               del_relation(<relation>=e <ancestor>=t1, <descendant>=t2);}}}}
}
```

ALGORITHM 3. Subnet for the priority relation.

events is modeled in the same way as in the case of the non-deterministic event dispatcher.

4.4. Hierarchy of states and transitions

One important feature of statecharts is the hierarchical structure of states. A state of a statechart can be a basic state (containing no other states), an OR state (containing only substates being active alternatively if the state itself is active) or an AND state (containing only concurrent regions).

Recall that 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 the firing of one of them inhibits the firing of the other, that is the intersection of the two sets of states they exit is not empty.

Transitions originating from substates of the source state of a transition take priority over the transition in question. When several transitions are enabled, their maximal non-conflicting set (with maximal priority) may fire at the same time in a single step. The priority relation defines a partial ordering relation over the set of transitions (because some transitions may be independent). Partial ordering relations are usually represented as tree structures.

The priority relation of transitions is used by the transformation. The transitions triggered by the same event are arranged in a priority tree according to the hierarchy of the transitions. (Trees are depicted as having a root at the top and leaves at the bottom; thus the directions 'up' and 'down' have to be understood accordingly.) On a path from the root to a leaf, a transition with higher priority is located closer to the leaf. Non-conflicting transitions and conflicting ones with equal priorities are located on different arcs of the priority tree. Compound transitions are mapped to a set of simple transitions.

The relation can be built as shown in Algorithm 3.

Figure 3 shows a small statechart as an example. Both state hierarchy (with A as a superstate of AA, AB, etc.) and concurrent regions (of state AB) are included. Eight transitions (*a* to *g*) are presented, all of which are triggered by the same event. (Transitions triggered by other events are not depicted.) The priority tree corresponding to the priority structure of the transitions is shown at the bottom of the figure. It consists of two types of nodes.

- Simple nodes (shown by nodes labeled with the names of UML transitions in Figure 3) represent the SRN subnets which correspond to the UML transitions. For the sake of simplicity, UML transitions are named in this figure by Greek letters and a special notation is used to distinguish these names from events (which are not shown, since recall that each transition in this figure has the same trigger event).
- Joining nodes (shown by empty circles) connect the subtrees.

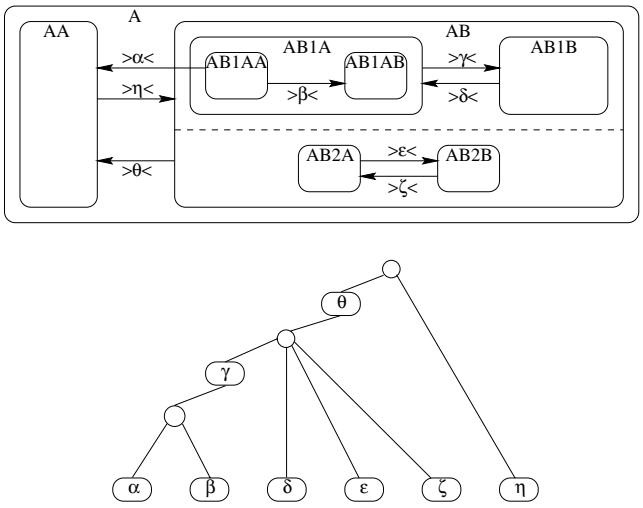

FIGURE 3. The tree structure of the priority relation.

The priority tree structure can be considered as a tree-like daisy chain of transitions. 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 joining nodes. 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 above, 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 subnets (corresponding to the simple and joining nodes of the priority tree) with an auxiliary control structure. This control structure consists 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

```
function set_of_predecessors(transition t) {
   set pred=empty;
   for (each state s of the SC) {
      if (((s is superstate of source(t) OR s is source(t)) AND
           s is not superstate of stable_target(t) AND s is not stable_target(t))
       OR (isRegion(s) AND
           parentstate(s) is superstate of source(t) AND parentstate(s) is not superstate of stable_target(t))) {
         put(<set>=pred, <element>=s);
      }
   }
   return pred;
}

function set_of_successors(transition t) {
   set succ=empty;
   for (each state s of the SC) {
      if ((s is superstate of stable_target(t) OR s is stable_target(t)) AND
          (s is not superstate of source(t) AND s is not source(t))) {
         put(<set>=succ, <element>=s);}}
   return succ;
}

function least_common_region(transition t) {
   state lcr=null;
   for (each state s of the SC) {
      if (isRegion(s) AND
          s is superstate of source(t) AND s is superstate of the stable_target(t) AND
          lcr is superstate of s) {
         lcr=s;
      }
   }
}

procedure build_nodes(transition t) {
   add_place(<name>="uncons_"+t);
   add_place(<name>="cons_"+t);
   add_place(<name>="still_uncons_"+t);
   add_trans(<name>=t+"_yes", <type>=timed, <param>=(tagged value "rate" of the UML transition t),
             <guard>='(AND(not_empty(p) for each place p, the name of which is in set_of_predecessors(t)))
                      AND (the guard of the UML transition t is true)');
   add_trans(<name>=t+"_not", <type>=immediate,
             <guard>='NOT(guard(t+"_yes")) OR
                      (OR(empty(p+"_n") for each place p, the name of which is in set_of_predecessors(t)))');
   add_iarc(<tr_name>=t+"_not", <pl_name>="uncons_"+t);
   add_iarc(<tr_name>=t+"_yes", <pl_name>="uncons_"+t);
   add_oarc(<tr_name>=t+"_not", <pl_name>="still_uncons_"+t);
   add_oarc(<tr_name>=t+"_yes", <pl_name>="cons_"+t);
   for (event e in sent_event(t)) {
      add_oarc(<tr_name>=t+"_yes", <pl_name>=e+"0");}
   for (each state s in enumerate_predecessors(t)) {
      add_iarc(<tr_name>=t+"_yes", <pl_name>=s+"_n"); }
   for (each state s in enumerate_successors(t)) {
      add_oarc(<tr_name>=t+"_yes", <pl_name>=s+"_n");}
   if (least_common_region(t)<>null) {
      add_iarc(<tr_name>=t+"yes", <pl_name>=least_common_region(t)+"_n");}
}
```

ALGORITHM 4. Subnet for a UML transition.

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.

4.4.1. Simple nodes

The formal description of building a simple node is presented by Algorithm 4, where `stable_target(t)` returns the set of basic states becoming active when transition t is firing. This set is formed by the target state of the transition and the following states, recursively:

- the initial substates of the target state;
- the initial states of the regions of the target state;
- the initial states of another region(s) of the superstate of the target state.

Figure 4b shows the SRN pattern of a simple node of the tree, belonging to the UML transition presented in Figure 4a. The UML transition is represented by the SRN transition *t1_yes*. The other SRN places represent the following items.

- Predecessor states. They are the states to be left when the UML transition fires (in this case it is only the state *DS*). The predecessor states are the source state of the transition and all of its superstates which are not superstates of the target states. The parallel regions of the superstate regions of the source state which are not superstate regions of the target states are also to be left. They can be identified according to the static structure of the statechart. Here both the predecessor and target state are substates of the common state *DC*.
  There could also be other states to be left, namely the active states of parallel regions of the predecessor

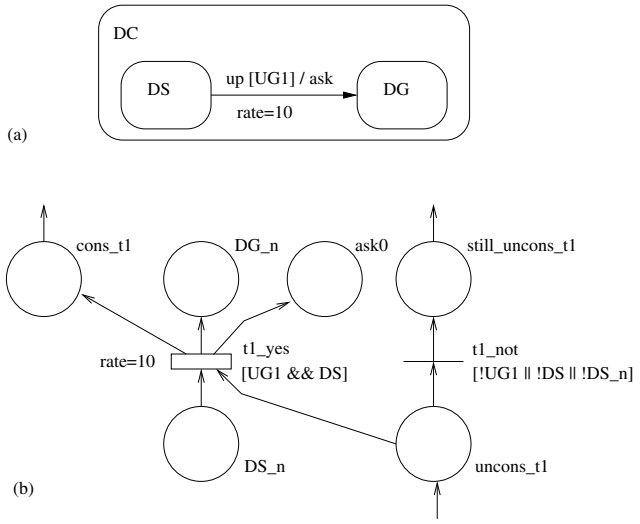

FIGURE 4. A simple UML transition (a) and the corresponding SRN pattern (b).

states. These states cannot be identified unambiguously by the static analysis of the statechart, thus exiting these states necessitates another solution (described later).
When the SRN pattern is generated, whether the corresponding UML transition is located within any concurrent region of the statechart has to be checked. The *least common region* (LCR) of a UML transition is characterized as follows:

- it is a region (direct substate of a concurrent state);
- it is the superstate of both the source and the target states;
- there is no lower region in the state hierarchy which fulfills the two previous criteria.

When the SRN transition corresponding to a UML transition fires, the token also has to be removed from the SRN place representing its LCR (if any). Without removing this token, in special situations the non-deterministic choice among conflicting UML transitions can not be assured. Namely, the conflict between a transition inside a region and another transition (from inside a concurrent region) to a target state outside of the parent state of these regions can not be detected. This token can be restored in the synchronization phase (see later) unambiguously.

- Successor states. These are the states to be entered when the transition fires (in this case it is the single state *DG*). This set of states can be unambiguously identified by analyzing the static structure of the statechart.
- The chain of unconsumed events. 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 appear in the chain of unconsumed events on the arcs of the appropriate priority tree structure of the triggered transitions. In Figure 4, places *uncons_t1* and *still_uncons_t1* are in this chain.
- The chain of consumed events. The token representing the event will be moved from the chain of unconsumed events to the chain of consumed events (here place *cons_t1*) if the transition *t1_yes* fires. If *t1_yes* cannot fire, *t1_not* fires, putting the token to place *still_uncons_t1*, i.e. the event remains unconsumed. (The guard of the transition *t1_not* expresses that *t1_yes* cannot fire.)
- Generation of events. This is implemented by outarc(s) from the timed SRN transition to the appropriate place(s) of the event dispatcher. Here in the example a token is passed to the place *ask0*.

The guard `[UG1 && DS]` of the SRN transition *t1_yes* contains the expression *UG1* (belonging to the guard of the UML transition) and *DS* that refers to the source state. Note first that the guard of this transition refers to *DS*, while it is connected to *DS_n* (and *DG_n*). The distinction between the input/output places of the transition (the 'next' places) and the places referred to in its guard (the 'last' places) will be described in detail in Section 4.6. Second, checking of the marking of the place *DS* is necessary to avoid firing when a token is generated to a 'next' place by another transition.

In this example the simplest timing policy was chosen, where the fastest of the enabled conflicting transitions fires. There are other possible policies as well, some of them are described in Section 4.5.

If there are two conflicting transitions of the statechart enabled at the same time then the firing of the corresponding SRN transitions occurs as follows.

- If one of them has higher priority, 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 the transition with higher priority.
- If they have the same priority or they are not related by the priority relation, 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. Two conflicting transitions cannot fire in the same step, because the one that fires first removes the token from the 'next' place representing the common parent state to leave. If two transitions have no common state to leave, they are not conflicting.

There is a special case in which a transition leaves and then enters the same (set of) states again, in the same step. This situation may also involve states in active parallel regions. Normally, the set of states left by a transition is handled by the so-called synchronization step (Section 4.6). However, in this special case whether a state was not affected by the firing of a transition or it was left and entered again cannot be distinguished. Accordingly, the source states have to be left explicitly by generating transitions for each possible state configuration to be left. This construction can be applied to each transition of a statechart and it does not increase the state space of the SRN (but involves more SRN transitions).

```
procedure build_joining_nodes(transition t, event e) {
   if (there is no transition t* with in_relation(<relation>=e, <ancestor>=t, <descendant>=t*)){
      add_oarc(<tr_name>="split_"+e, <pl_name>="uncons_"+t);}
   if (there is exactly one transition t* with in_relation(<relation>=e, <ancestor>=t, <descendant>=t*)){
      add_trans(<name>="forward_consumed_to_"+t, <type>=immediate);
      add_iarc(<tr_name>="forward_consumed_to_"+t, <pl_name>="cons_"+t*);
      add_oarc(<tr_name>="forward_consumed_to_"+t, <pl_name>="cons_"+t);
      add_trans(<name>="forward_unconsumed_to_"+t, <type>=immediate);
      add_iarc(<tr_name>="forward_unconsumed_to_"+t, <pl_name>="still_uncons_"+t*);
      add_oarc(<tr_name>="forward_unconsumed_to_"+t, <pl_name>="uncons_"+t);}
   if (there are more transitions t* with in_relation(<relation>=e, <ancestor>=t, <descendant>=t*)){
      n = {number of such transitions t*}
      add_place(<name>="waiting_to_"+t);
      add_trans(<name>="forward_consumed_to_"+t, <type>=immediate);
      add_iarc(<tr_name>="forward_consumed_to_"+t, <pl_name>="waiting_to_"+t, <multiplicity>=n);
      add_oarc(<tr_name>="forward_consumed_to_"+t, <pl_name>="cons_"+t);
      add_trans(<name>="forward_unconsumed_to_"+t, <type>=immediate);
      add_oarc(<tr_name>="forward_unconsumed_to_"+t, <pl_name>="uncons_"+t);
      for (each transition t* with in_relation(<relation>=e, <ancestor>=t, <descendant>=t*)) {
         add_iarc(<tr_name>="forward_unconsumed_to_"+t, <pl_name>="uncons_"+t*);
         add_trans(<name>="collect_consumed_from"+t*, <type>=immediate);
         add_iarc(<tr_name>="collect_consumed_from"+t*, <pl_name>="cons_"+t*);
         add_oarc(<tr_name>="collect_consumed_from"+t*, <pl_name>="waiting_to_"+t);
         add_trans(<name>="collect_unconsumed_from"+t*, <type>=immediate, <guard>={"waiting_to_"+t is not empty});
         add_iarc(<tr_name>="collect_unconsumed_from"+t*, <pl_name>="still_uncons_"+t*);
         add_oarc(<tr_name>="collect_unconsumed_from"+t*, <pl_name>="waiting_to_"+t);}}
}
```

ALGORITHM 5. Subnet for a joining node in the tree structure.

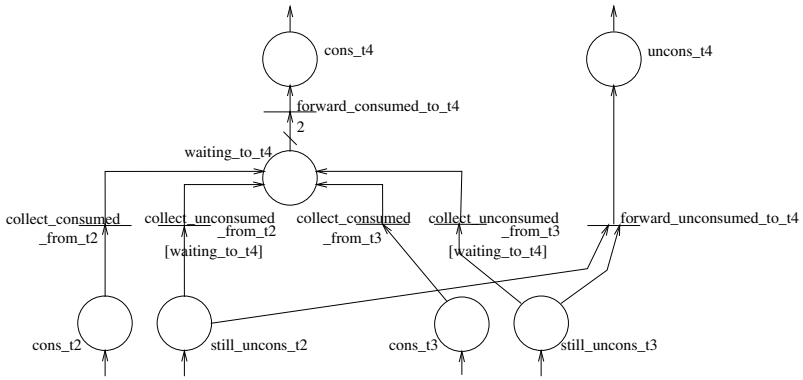

FIGURE 5. SRN pattern of a joining node in the tree structure.

4.4.2. Joining nodes

A joining node of the tree merges the event chains of its subtrees (Algorithm 5). An example subnet is depicted in Figure 5. 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 the event is unconsumed in this common path if and only if the event was not consumed by any of the transitions of the subtree.

The event is 'consumed' in the common path when some of the transitions of the subtree have already fired (they have carried over the tokens on the 'consumed' chain) and the other transitions cannot fire (they have passed on the tokens along the chain). This construction ensures that when the token representing the event reaches the root of the tree, then no more sub-SRNs corresponding to transitions of the statechart will fire, thus the step has to be finished.

In our example the two joining arcs are represented by the two place pairs *consumed_t2*, *still_unconsumed_t2* and *consumed_t3*, *still_unconsumed_t3*. According to the previous pattern (Figure 4), one token can be found either in place *consumed_t2* or in place *still_unconsumed_t2*, and another token can be found either in place *consumed_t3* or in place *still_unconsumed_t3*.

If the event was not consumed by the transitions on the joining arcs, then there are tokens in places *still_unconsumed_t2* and *still_unconsumed_t3*. In this case transition *forward_unconsumed_to_t4* can fire, and the control is passed to a transition on the common (joined) arc with lower priority (here a token is put to place *unconsumed_t4*) or, if there are no transitions with lower priority, a token is put to the place *READY* belonging to the dispatcher (Figure 1).

If the event was consumed by one or both of the transitions on the joining arcs, then there is a token in place *consumed_t2* or/and in place *consumed_t3*. Thus, transition *collect_consumed_from_t2* or/and *collect_consumed_from_t3* can fire. Token(s) will be put to the place *waiting_to_t4*, which may enable the token from the place representing an unconsumed event (if any) to be removed. If there are as many tokens in place *waiting_to_t4* as the number of arcs to be joined (here 2), then transition *forward_consumed_to_t4* will fire and a token appears in the place *consumed_t4*

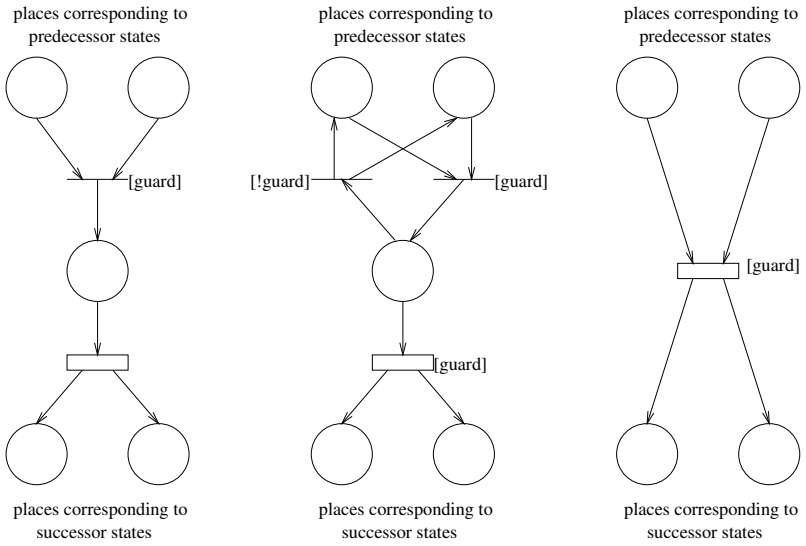

FIGURE 6. Models for combining guards and timing.

representing on the common arc that the event was already consumed.

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

4.5. Time semantics

The relationship between timing and guard evaluation is not specified in standard UML. In our approach, time delay is associated with UML transitions, assuming that this delay is due to program code execution, communication delay or fault activation. 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 is a better fit to the majority of practical problems.

4.5.1. Guarded transitions

We describe three possible semantics for timed and guarded UML transitions and their transformation patterns. They may fit to different applications. The three alternatives are as follows (Figure 6 shows the implementations):

- the selection of the transitions is irrespective of timing constraints (a);
- the guard has to be true permanently during the delay or else the transition will be deselected (b);
- the 'fastest' of the enabled transitions wins (c) (this is the one used in the example in this paper).

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 the fact that the transition can only fire if the appropriate state was active before the actual step. Figure 6a–c shows sub-SRNs corresponding to the transitions of the statechart.

The types and parameters of the timed SRN transitions correspond to the types and parameters of the corresponding statechart transitions. The timing policy (resampling, race with age/enabling memory, etc.) is determined by the designer and must be implemented by the SRN tool used for the analysis.

In Algorithm 4 the third alternative (Figure 6(c)) is implemented, but any of the alternatives mentioned here can be used by changing `transition(t+"_yes")` and its environment appropriately.

4.6. Step semantics

The UML semantics requires the evaluation of the guards of the transitions at the beginning of a step, before the firing of any transition. 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 at 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 other transitions are evaluated. For 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 SRN places representing the states of the statechart are duplicated. For a state A there is an SRN place A containing a token if and only if the state A was active just before the actual step (called the *last* place in the following) and there is another SRN place A_n containing a

```
function put_in_states(state s) {
   state s_last=s;
   for (each substate s* of s) {
      add_relation(<relation>="S",
                   <fore>=s_last, <back>=s*);
      s_last=put_in_states(s*);}
   return(s_last);
}

procedure generate_list_of_states() {
   new_relation(<name>=S,
                <type>=unidirectional,
                <fields>=(fore, back));
   put_in_states({top state of the SC});
}
```

ALGORITHM 6. Subnet for generating a list of places.

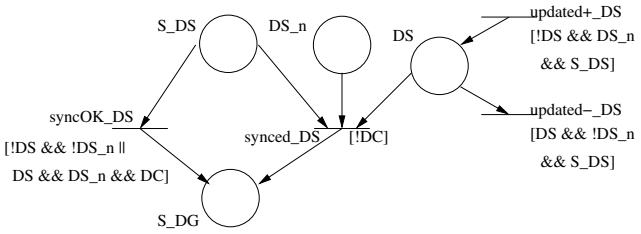

FIGURE 7. Synchronization of the duplicated places.

token if and only if the state *A* will be active after the actual step (called the *next* place in the following).

The SRN places *DS_n* and *DG_n* in Figure 4 depict the *next* places, while the guards of the appropriate SRN transitions in the subnet are expressions over the marking of the SRN 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. In the tree structure of the triggered transitions, when the token representing the selected event reaches the root of the tree, it is passed to a *synchronization chain*. All states of the statechart 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 statechart states. The synchronization of the duplicated places could happen independently, but the non-deterministic ordering would increase the state space of the SRN without any advantage. A fixed ordering avoids this kind of state space explosion.

A depth-first tour on the graph of the state hierarchy is implemented in Algorithm 6. The pattern for the synchronization is shown in Algorithm 7.

Figure 7 depicts the synchronization pattern of the UML state *DS*, where the SRN places *DS* and *DS_n* are synchronized. There is a token in *DS* if and only if the state *DS* of the statechart was active just before the actual step, and there is a token in place *DS_n* if and only if the state *DS* of the statechart will be active after the actual step. The SRN place *DC* (not shown in the figure but referred to in guards) represents the direct parent state of (the UML state) *DS*. The places *S_DS* and *S_DG* are two places in the synchronization chain. A token is passed from *S_DS* to *S_DG* (for synchronizing the next state in the order of the synchronization chain) if *DS* and *DS_n* are already synchronized by the transitions on the right-hand side of the figure or the places are cleared when the parent state of *DS* is not active.

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 a statechart transition fires, the tokens must be removed from places representing states considered to be inconsistently active, since their parent states are inactive. Remember that the predecessor states on Figure 4 are only the source and parent states of the statechart transition and the regions parallel to them, which are to be exited. However, there may be 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. Also recall that the places corresponding to the LCR of the transitions were emptied as well and these missing tokens have to be put back to those places. The places with missing tokens can be identified unambiguously, because no regions of the statechart can be inactive if their parent states are active. Note that these transient problems do not affect the result of the step.

For example, in Figure 3 the predecessor states of the transition *a* are *AB1AA*, *AB1A*, *AB1* and *AB*. If *a* is enabled then either *AB2A* or *AB2B* must be active (since their parent state *AB2* is active). Which of them is active at the given situation cannot be identified statically, therefore they do not appear in the set of predecessor states of *a*. Before the end of the step when *a* fires, the one that is active must be exited, because their parent state *AB* was exited.

4.7. Composition of subnets

The composition of the subnets includes three tasks as shown in Algorithm 8.

- A single SRN place 'READY' is generated that represents the end of a step.
- Three places (the 'last', 'next' and 'synchronization' places) are generated for each state of the statechart. If a given state is active in the initial state configuration of the statechart then the 'last' and 'next' places are marked.
- The subnets described in the previous sections are generated and connected by additional arcs.

The necessary number of patterns is as follows.

- The number of event queues and the type of event dispatcher(s) are defined by the designer (additional information is attached to the UML model). Global event dispatching, event dispatching per object, event dispatching per statechart and FIFO or non-deterministic dispatching can be selected.

```
   procedure build_synchronization(state s) {
      state par_s={the parent state of s in the SC};
      add_trans(<name>="syncOK_"+s, <type>=immediate,
                <guard>='(s and s+"_n" are both empty) OR (neither s nor s+"_n" nor parentstate(s) are empty)');
      add_iarc(<tr_name>="syncOK_"+s, <pl_name>="S_"+s);
      add_trans(<name>="synced_"+s, <type>=immediate, <guard>='parentstate(s) is empty');
      add_iarc(<tr_name>="synced_"+s, <pl_name>="S_"+s);
      add_iarc(<tr_name>="synced_"+s, <pl_name>=s+"_n");
      add_iarc(<tr_name>="synced_"+s, <pl_name>=s);
      add_trans(<name>="updated+_"+s, <type>=immediate, <guard>='s is empty but neither s+"_n" nor "S_"+s');
      add_oarc(<tr_name>="updated+_"+s, <pl_name>=s);
      if (isRegion(s)) {
         add_trans(<name>="updated-_"+s, <type>=immediate,
                   <guard>='s and "S_"+s are not empty but s+"_n" and parent(s) do');
         add_trans(<name>="restoreregion_"+s, <type>=immediate,
                   <guard>='s, "S_"+s and parent(s) are not empty but s+"_n" does');
         add_oarc(<tr_name>="restoreregion_"+s, <pl_name>=s+"_n");
      }
      else {
         add_trans(<name>="updated-_"+s, <type>=immediate,
                   <guard>='s and "S_"+s are not empty but s+"_n" does');
      }
      add_iarc(<tr_name>="updated-_"+s, <pl_name>=s);
      if (there is a state s* with in_relation(<relation>="S", <fore>=s, <back>=s*)) {
         add_oarc(<tr_name>="syncOK_"+s, <pl_name>="S_"+s*);
         add_oarc(<tr_name>="synced_"+s, <pl_name>="S_"+s*);}
      else {
         add_oarc(<tr_name>="syncOK_"+s, <pl_name>="READY");
         add_oarc(<tr_name>="synced_"+s, <pl_name>="READY");}
   }
```

ALGORITHM 7. Subnet for synchronization of the duplicated places.

```
procedure SC2SRN() {
   add_place(<name>="READY", <init>=1);

   for (each state s of the SC) {
      if (s in {initial configuration of the SC}){
         add_place(<name>=s, <init>=1); add_place(<name>=s+"_n", <init>=1); add_place(<name>="S_"+s);}
      else {
         add_place(<name>=s); add_place(<name>=s+"_n"); add_place(<name>="S_"+s);}}

   if (event dispatching is non-deterministic){
      for (each event e of the SC) {build_nondet_dispatcher(e);}
   else {
      n = ((length of the queue)-1);
      for (i=0 to n) {add_place(<name>="queue_"+i);}
      for (each event e of the SC) {build_fifo_dispatcher(e);}

   for (each transition t of the SC) {build_nodes(t);}

   for (each event e of the SC) {
      generate_tree(e);

      for (each transition t of the SC) {build_joining_nodes(t, e);}
      n = (number of transitions t, which have no transitions t* with
           in_relation(<relation>=e, <ancestor>=t*, <descendant>=t));
      add_place(<name>="waiting_to_top");
      add_trans(<name>="forward_consumed_to_top", <type>=immediate);
      add_iarc(<tr_name>="forward_consumed_to_top", <pl_name>="waiting_to_top", <multiplicity>=n);
      add_oarc(<tr_name>="forward_consumed_to_top", <pl_name>="S_"+{top state of the SC});
      add_trans(<name>="forward_unconsumed_to_top", <type>=immediate);
      add_oarc(<tr_name>="forward_unconsumed_to_top", <pl_name>="READY");
      for (each transition t, which has no transitions t* with
           in_relation(<relation>=e, <ancestor>=t*, <descendant>=t)) {
         add_iarc(<tr_name>="forward_unconsumed_to_top", <pl_name>="uncons_"+t);
         add_trans(<name>="collect_consumed_from"+t, <type>=immediate);
         add_iarc(<tr_name>="collect_consumed_from"+t, <pl_name>="cons_"+t*);
         add_oarc(<tr_name>="collect_consumed_from"+t, <pl_name>="waiting_to_top");
         add_trans(<name>="collect_unconsumed_from"+t, <type>=immediate, <guard>={"waiting_to_top" is not empty});
         add_iarc(<tr_name>="collect_unconsumed_from"+t, <pl_name>="still_uncons_"+t);
         add_oarc(<tr_name>="collect_unconsumed_from"+t*,<pl_name>="waiting_to_top");}}
   generate_list_of_states();
   for (each state s of the SC) {build_synchronization(s);}
}
```

ALGORITHM 8. The main procedure to compose the subnets.

- There are as many transition hierarchy trees as the number of events handled by the transitions of the statecharts of each event dispatcher.
- 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 statechart has to be mapped to the SRN by inserting tokens into the corresponding pairs of places. The initial marking of the place READY has to be 1.

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. MODELING OF FAULTS

In this section how faults and errors can be modeled is shown. Explicit modeling of component faults and identification of system failures will allow us to compute reliability and availability figures.

The following types and locations of a fault can be distinguished. Design faults can exist in hardware and software. (In fact the co-design paradigm is gradually making hardware and software indistinguishable.) Certain physical faults occur inside a single component of the system and can be handled by that component. Some physical faults occur inside a component but must be handled by another component. External faults occur in the environment and are often transient. Faults can give rise to errors, that is to undesired system states, which in turn can lead to the failure of the system [18].

Augmenting the system model with a realistic fault model is the basis for the dependability analysis. Faults are modeled, for instance, by message losses or loss of synchrony. Errors can be modeled by so-called state perturbations. State perturbations include distinguished states corresponding to degraded performance of the modeled system, paths leading to such states, erroneous state transitions, trigger events due to external faults giving rise to erroneous state transitions and the use of guards to express fault-tree-like failure conditions. Thus, a wide spectrum of possible errors can be modeled.

Our error model is based on the notion of state perturbations. For example, unintended state transitions are state perturbations. An unintended transition from state s to state q may be due to a permanent or temporary fault and q may be an erroneous state. An unintended state transition due to a temporary fault occurs at most once in the considered period. An unintended state transition caused by a permanent fault can occur whenever the system is in the state that gives rise to the erroneous transition.

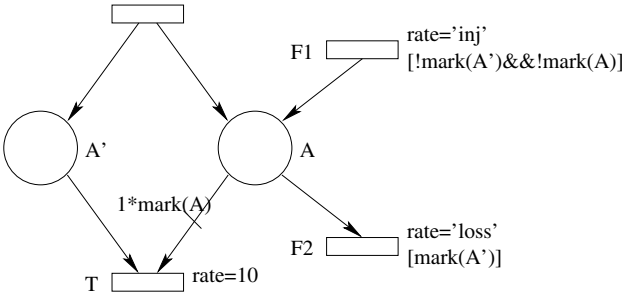

FIGURE 8. Modeling of corrupt signals.

Such state perturbations can be modeled by binary and reflexive relations over the state space of a statechart [19, 20, 21, 22].

Signal losses can mean that events or in-state guards are not observed. The trigger event is lost or the guard always evaluates to TRUE. In this way, sensor and actuator faults or the loss of messages can also easily be modeled.

Finally, guards can express dependability requirements in the form of negations of fault trees (Boolean expressions) over component states. For instance, a fault tree defining possible collisions of certain devices that could lead to the failure can be specified. In this way, dependability requirements, resulting from the requirement analysis, can be directly integrated into the system model.

As mentioned, our fault model includes corrupted actuator and sensor signals. Besides modeling the loss, duplication or corruption of events (spurious events), a guard can also sense an active signal state as being inactive and *vice versa*. In this case we duplicate the places corresponding to signal states (Figure 8). Place A' models the state of the signal and place A models the presence of the signal (public state). A fault occurs when places A' and A have different markings (see below). The arc annotation $1*mark(\dots)$ defines a state-dependent capacity of the arc. For example, if $mark(A) = 0$, then firing of the output transition T depends only on the marking of place A'.

Two normal and two faulty cases can be distinguished:

1. both places are empty, the transition T cannot fire;
2. both places contain tokens, the transition T can fire;
3. only A' contains a token, i.e. the fault 'signal is lost' occurred. Then the transition T can fire;
4. only A contains a token, i.e. a fault 'spurious signal' occurred, then the transition T cannot fire. However, the guards of other transitions (which refer to this public state) evaluate to TRUE.

The faults are modeled as the results of the firing of transitions *F1* and *F2*. The modeler has only to specify the corresponding failure (firing) rates *'inj'* and *'loss'*.

6. MODEL ANALYSIS

The SRN (generated from the UML model according to the patterns described in the previous sections) can be analyzed by the SRN tools PANDA or SPNP. In certain cases

(in particular, when transitions have exponential firing times) analytic solutions are possible, otherwise simulations have to be performed. Both tools support the computation of steady-state measures (if a steady state exists) and transient analysis.

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

- the reachable 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 can be derived directly from the results presented above (throughput, utilization). In more complex cases, user-defined reward functions can also be used.

Dependability-based analysis in this framework requires the explicit modeling of faulty behavior and the explicit identification of erroneous states, as presented in the previous section. The analysis of the probability of erroneous states leads to reliability (if no repair is modeled) and availability characteristics (if a repair is modeled). Analogously, safety figures can be derived by distinguishing the unsafe states in the model. Other application-specific measures may combine the performance characteristics with fault modeling (e.g. the performance of the system in the case of an error, utilization of a repair facility, etc.). This means to specify non-functional requirements in terms of structured English sentences with its syntax is presented in [23], by clear and consistent notation. The translation steps for interpretation and verification of these sentences are also shown in the same paper.

The analysis of detailed GSC and statechart models is very time consuming and needs high-performance computers. Full models of realistic applications usually have higher complexity than modern tools and computers can handle. Thus, quantitative analysis should be focused on certain system components such as core parts of the embedded controllers. They can be modeled in detail, while the other system components need not be modeled in such depth. Here the connection with the system-level structural dependability analysis [24] could be important: system-level sensitivity analysis can identify critical components, while the analysis of dynamic behavior provides parameters useful in the computation of (system-level) dependability attributes.

Another way to reduce complexity is to deduce certain scenarios from the statechart model and model them by sequence diagrams. Usually these sequence diagrams are much less complex than the statechart model itself. The transformation of sequence diagrams to SRNs has also been elaborated. Characteristics like runtime and termination probability of selected scenarios can be computed by the SRN tools [6].

7. APPLICATION OF THE TRANSFORMATION

The transformations described in this paper have already been applied to several examples. We can mention, among others, the 'Trajectory Planner' example of a spacecraft [25] and the model of the replication manager in distributed object-oriented software [26]. In the latter case, the possibility of modeling and analyzing systems with event processing was necessary in order to be able to examine the fault notification mechanism, error processing and recovery techniques and their effects on system availability.

Here we illustrate our approach by an example of an embedded fault-tolerant system, a variation of a production cell model [27, 28], because it is a well-known benchmark example for the modeling of distributed embedded systems and we can present both the GSC and UML statechart models.

The system contains a press that processes metal plates, a robot with an extensible arm (with an electromagnet) for loading and unloading the press and a repair console. The feed belt as well as the deposit belt are not modeled explicitly. The breakdown of the press can be sensed by the repair console, so that the repairman (worker) can repair the press. The robot arm may also become stuck and will also then be repaired by the repairman.

The complete UML model of an extended version of this example is given in [29]. It comprises a requirement model, an object model and a deployment model.

The dynamic view of the system is given by statecharts. According to our modeling approach, each device model consists of a hardware behavioral model and the statechart of the corresponding controller (a single, central cell controller or that of several distributed device controllers). In the first phase of the modeling, we used the higher-level GSC diagrams; later in the refinement phase full statecharts were prepared.

The complete GSC model comprises five statecharts (with nine state transition diagrams and 34 basic states, of which eight are sensor states and eight are actuator states). The GSC model of the press (Figure 9) consists of two components, one for the hardware of the press and one for its controlling unit. This part of the model contains two sensor and four actuator states. (The guards of some transitions on the figure apply to states of other components not presented here.) A possible malfunction of the press hardware is modeled as a kind of state perturbation, which can be detected by the controlling unit. For the sake of simplicity, the transitions of the reparation are omitted.

The full statechart model of the same system consists of one single hierarchical statechart with 15 concurrent states containing 50 substates and 68 transitions triggered by 42 events (14 timer events). A single global event queue is assumed, 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. To illustrate the model, the statechart corresponding to the hardware of the rotary table is depicted in Figure 10.

For the quantitative analysis of the models, the SRN tool PANDA was used. The transformed GSC model (as the

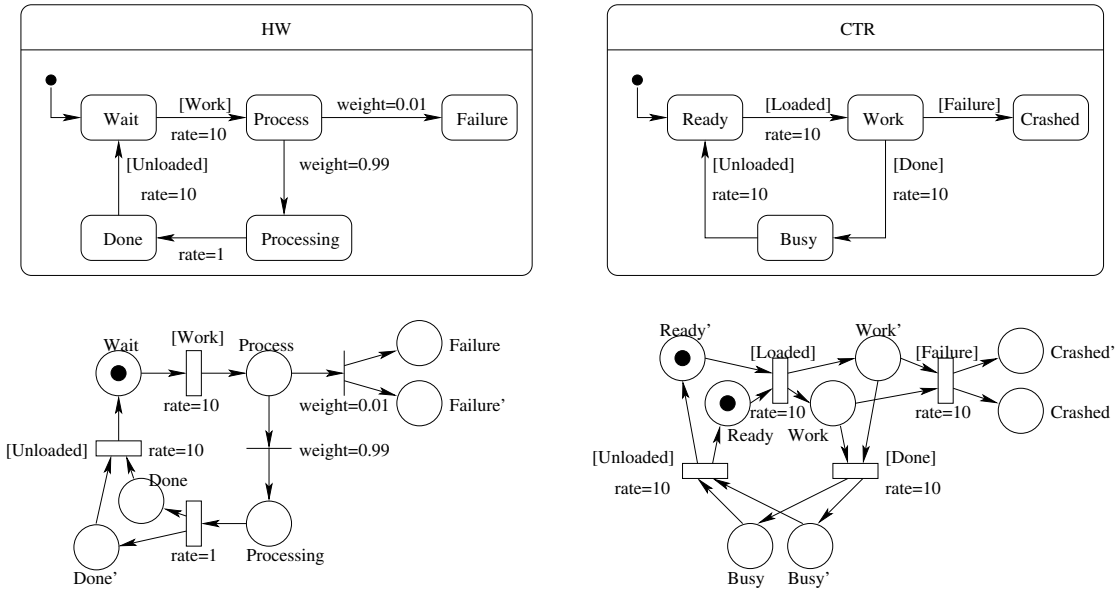

FIGURE 9. GSC model and the corresponding SRN model of the press.

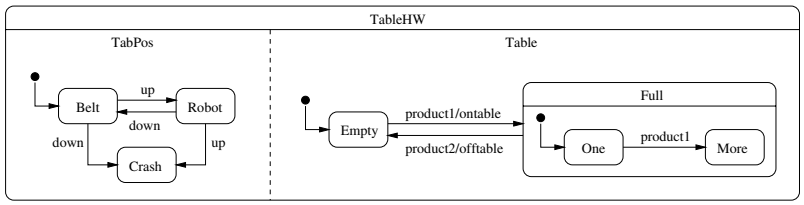

FIGURE 10. Statechart model of the table (hardware).

components are strongly coupled by the guards) has 9316 reachable states. The size of the state space of the full statechart model increases if a FIFO dispatching policy is selected; the increase depends heavily on the length of the queue.

Various performance and dependability results were computed by PANDA [6]. For example, computing the utilization of the repairman as a function of the elapsed time shows that the utilization increases to 0.15. The throughput of the system (the mean number of forged plates per time unit) was also computed as a function of the signal loss rate. There is a domain of the signal loss rate between 10 and 1000 where the throughput is particularly sensitive to the loss rate (the throughput rapidly decreases to 20%).

Failure events like the breakdown of the robot arm and its repair were analyzed as special scenarios. The distribution function of the time to load the press after the breakdown shows that on average 64 s is required. Another experiment compared the fault-free case and the scenario when the signal from the robot control was lost twice. The average duration increased by 33%.

8. CONCLUSION

We have presented a method which allows quantitative dependability and performance analysis of systems modeled by the use of UML statechart diagrams. To facilitate the trade-off between the details of modeling and the complexity of the analysis, both the higher-level, simplified formalism (GSC) and the full UML statecharts were supported by the transformation and the corresponding analysis.

Our transformation from UML statecharts to SRNs covers a large subset of model elements including event processing, state hierarchy and transition priorities. The transformations were presented in the form of transformation patterns. The properties of the resulting SRNs 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 number of states generated (the state space of the underlying Markov chain) corresponds to the number of state configurations of the UML model.

By the transformation, the possibility of using UML to model and analyze error-prone and fault-tolerant system behavior is greatly enhanced. In the case of complex systems this kind of analysis should be restricted to core critical parts of the system, such as a (central) controller or a replication manager, since the analysis is based on a detailed model of the system.

ACKNOWLEDGEMENTS

This work was partially supported by the projects ESPRIT Open LTR 27439 ‘HIDE’, the Hungarian-German

Researchers Exchange Program No. 8, and OTKA-F030553 and OTKA-T30804 (Hungarian NSF). The authors would like to thank the reviewers for their constructive comments.

REFERENCES

[1] Rumbaugh, J., Jacobson, I. and Booch, G. (1999) *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, Inc., Reading, MA.

[2] OMG (1997) *UML Semantics, Version 1.3*. Object Management Group, Needham, MA. http://www.omg.org.

[3] Bondavalli, A., Dal Cin, M., Latella, D. and Pataricza, A. (1999) High-level integrated design environment for dependability (HIDE). In *Proc. Fifth Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-99F)*, Monterey, CA, November 18–20, pp. 87–92. IEEE Computer Society, Los Alamitos, CA.

[4] Harel, D. (1987) Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, **8**, 231–274.

[5] Dal Cin, M., Huszerl, G. and Kosmidis, K. (1999) Transformation of guarded statecharts for quantitative evaluation of dependable embedded systems. In *Proc. 10th Eur. Workshop on Dependable Computing (EWDC-10)*, Vienna, Austria, May 6–7, pp. 143–148. Österreichische Computer Gesellschaft, Vienna.

[6] Dal Cin, M., Huszerl, G. and Kosmidis, K. (1999) Quantitative evaluation of dependability critical systems based on guarded statechart models. In *Proc. HASE'99, Fourth IEEE Int. Symp. on High Assurance Systems Engineering*, Washington, DC, November 17–19. IEEE Computer Society, Los Alamitos, CA.

[7] Ajmone Marsan, M. (1991) Stochastic Petri nets: an elementary introduction. In Rozenberg, G. (ed.), *Advances in Petri Nets*, pp. 1–29. Springer, Wien, Berlin.

[8] Ciardo, G., Blakemore, A., Chimento, P., Muppala, J. and Trivedi, K. (1993) Automated generation and analysis of Markov reward models using stochastic reward nets. In Meyer, C. and Plemmons, R. J. (eds), *IMA Volumes in Mathematics and its Applications: Linear Algebra, Markov Chains and Queueing Models*, Vol. 48, pp. 145–191. Springer, Berlin.

[9] Muppala, J. K., Ciardo, G. and Trivedi, K. S. (1994) Stochastic reward nets for reliability prediction. *Commun. Reliab., Maintain. Service.*, **1**, 9–20.

[10] Bause, F., Buchholz, P. and Kemper, P. (1994) Hierarchically combined queueing Petri nets. In *Proc. 11th Int. Conf. on Analysis and Optimization of Systems, Discrete Event Systems*, Sophia-Antipolis, France, June. *Lecture Notes in Computer Science*, **199**, 176–182. Springer, Wien, New York.

[11] Bernardo, M. and Gorrieri, R. (1996) Extended Markovian process algebra. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Pisa, Italy, August 26–29, pp. 315–330. *Lecture Notes in Computer Science*, **1119**. Springer, Wien, New York.

[12] Donatelli, S., Hillston, J. and Ribaudo, M. (1995) 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, NC, October 3–6. IEEE Computer Society, Los Alamitos, CA.

[13] Ciardo, G., Muppala, J. and Trivedi K. S. (1989) SPNP—stochastic Petri net package. In *Proc. IEEE 3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, Kyoto, Japan, December 11–13, pp. 142–151. IEEE Computer Society, Los Alamitos, CA.

[14] Allmaier, S. and Dalibor, S. (1997) PANDA—Petri net ANalysis and Design Assistant. In *Tools Descriptions, 9th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation (Tools'97)*, St. Malo, France, June 2–6. *Lecture Notes in Computer Science*, **1245**, 58–60. Springer, Wien, New York.

[15] Douglass, B. P. (1998) *Real-Time UML*. Addison-Wesley Longman, Inc., Reading, MA.

[16] Latella, D., Majzik, I. and Massink, M. (1999) 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 Systems*, Florence, Italy, February, pp. 331–347. Kluwer, New York.

[17] Huszerl, G. and Majzik, I. (2000) Quantitative analysis of dependability critical systems based on UML statechart models. In *Proc. HASE 2000, Fifth IEEE Int. Symp. on High Assurance Systems Engineering*, Albuquerque, NM, November 15–17, pp. 83–92. IEEE Computer Society, Los Alamitos, CA.

[18] Lee, P. A. and Anderson, T. (1990) *Fault Tolerance, Principles and Practice*. Springer, Wien, New York.

[19] Dal Cin, M. (1998) Checking modification tolerance. In *Proc. Third IEEE Int. High-Assurance Systems Engineering Symp., HASE 98*, Washington, DC, November 13–14, pp. 9–12. IEEE Computer Society, Los Alamitos, CA.

[20] Dal Cin, M. (1997) Verifying fault-tolerant behavior of state machines. In *Proc. Second IEEE High-Assurance Systems Engineering Workshop, HASE 97*, Washington, DC, August 11–12, pp. 94–99. IEEE Computer Society, Los Alamitos, CA.

[21] Dal Cin, M. (1998) Modeling fault-tolerant system behavior. In Albrecht, R. (ed.), *Advances in Computing Science*, Vol. 3, pp. 213–234. Springer, Wien, New York.

[22] Huszerl, G. (1998) *Formal Verification of Fault-Tolerant Systems. A Relational Approach to Model Checking*. Master's Thesis, TU Budapest/University of Erlangen–Nuremberg.

[23] Dal Cin, M. (2000) Structured language for specification of quantitative requirements. In *Proc. HASE 2000, Fifth IEEE Int. Symp. on High Assurance Systems Engineering*, Albuquerque, NM, November 15–17, pp. 221–227. IEEE Computer Society, Los Alamitos, CA.

[24] Bondavalli, A., Majzik, I. and Mura, I. (1999) Automated dependability analysis of UML designs. In *Proc. 2nd IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, Saint Malo, France, May 2–5, pp. 139–144. IEEE Computer Society, Los Alamitos, CA.

[25] Huszerl, G. and Kosmidis, K. (2000) UML—extensions for quantitative analysis. In *Proc. UML 2000 Workshop: Dynamic Behaviour in UML Models: Semantic Questions*, York, October, pp. 70–75. LMU-München, Institut für Informatik, München, Germany.

[26] Huszerl, G. and Majzik, I. (2001) Modeling and analysis of redundancy management in distributed object-oriented systems by using UML statecharts. In *Proc. 27th Euromicro Conf., Workshop on Software Process and Product Improvement*, Warsaw, Poland, September 4–6, pp. 200–207. IEEE Computer Society, Los Alamitos, CA.

[27] Lewerentz, C. and Lindner, Th. (1995) *Formal Development of Reactive Systems: Case Study Production Cell*. *Lecture Notes in Computer Science*, **891**. Springer, Wien, New York.

[28] Matos, G., Purtilo, J. and White, E. (1997) Automated computation of decomposable synchronization conditions. In *Proc. Second IEEE High-Assurance Systems Engineering Symp., HASE 97*, Washington, DC, August 11–12, pp. 72–77. IEEE Computer Society, Los Alamitos, CA.

[29] Csertán, Gy., Dal Cin, M., Huszerl, G., Jávorszky, J., Kosmidis, K., Pataricza, A. and Szász, Cs. (1998) *The Demonstrator.* Technical Report ESPRIT Project 27439 (HIDE) deliverable 5 (HIDE/D5/TUB/1/v2). http://www.inf.mit.bme.hu/FTSRG/Publications/.