Quantitative Analysis of UML Statechart Models of Dependable Systems

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

[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 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 patterns. Performance measures can be directly derived using SRN tools, while dependability analysis requires explicit modeling of erroneous states and faulty behavior.

Keywords: UML statecharts, dependability analysis, performance analysis

Received November 1, 2000

1. INTRODUCTION

The role of formal modeling and analysis techniques in the development process of modern computer controlled systems becomes 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, production etc. not only the functional correctness, but also the reliability, availability, safety and performability have to be analyzed. 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, for instance, with error coverage, mean duration of a recovery cycle, the probability of tolerating certain state perturbations, or the probability of a failure. For such an analysis, additionally to the system's function, also the modeling of faults is necessary, including both the possible internal faults and the faults concerning the system's interaction with its environment (via sensors and actuators).

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) [RJB99] provides a visual notation (standardized by the Object Management Group [OMG97]) 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 the recent years, several methods were elaborated to enable us also the formal analysis of UML based designs. Among others, problems of system-level dependability modeling, formal verification, performance analysis of (subsets of) UML models were solved [BDLP99].

Our work is focused on the quantitative dependability and performance analysis of the UML behavioral models of embedded systems. The dynamic behavior of the system is given in UML by statechart diagrams [OMG97], an object-oriented mutation of classical Harel statecharts [Har87]. 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, when timing information is assigned to state transitions, and dependability analysis, if the model is extended with explicit failure states/events and probabilistic information. Although the UML notation has not been designed for these purposes, its standard mechanisms enable 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).

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 details in modeling and the degree of possible automation of the analysis. This lead us to define a sub-class of UML statecharts comprising so-called Guarded Statecharts (GSC) [DHK99b]. 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 [DHK99a]. 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, which concept may be of crucial importance in modeling the real architecture of the system. Moreover, this formalism prohibits the use of state hierarchy, one of the most useful concepts in statechart diagrams. Accordingly, we covered event processing and state hierarchy. Reaching the level of 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 will increase.

The analysis is based on a transformation from these statechart models to Petri nets 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 with firing time distributions of transitions, like Generalized Stochastic Petri Nets [AM91] and Stochastic Reward Nets [CBC$^+$92, MCT94], offer not only precise mathematical background but also sophisticated analysis tools. Although there are also other methods for quantitative analysis (like queuing networks [BBK94], stochastic process algebra [BG96] etc.) Petri nets are still considered to be the most mature in terms of the scope of theoretical results, the efficiency of the analysis algorithms and the number of available tools [DHR95]. 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.

The paper is structured as follows. The next section introduces the approach of UML-based model analysis and the design environment. In Section 3 the Guarded Statechart 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. Fault modeling, as a crucial step in dependability analysis, is discussed in Section 5. The application of the transformations in dependability and performance analysis is discussed in Section 6. An illustrative example is presented in Section 7. The paper is closed by the section of conclusions.

2. THE HIDE APPROACH

UML models are not directly amenable to quantitative analysis. Therefore, a method has to be introduced which generates a mathematical model that can be evaluated. The HIDE (High-Level Integrated Design Environment for Dependability[2]) project aimed at proposing a general answer to this need, integrating the design, validation and verification techniques through a *transformational approach* that targets the most common analysis tools [BDLP99]. The UML design was extended by using its standard mechanisms to include all the necessary details and parameters that are required to a quantitative analysis. Then the UML model was transformed automatically to the input formalism of the analysis tool. The results of the analysis were back-annotated to the original UML model highlighting design faults, bottlenecks and identifying to a certain level the possible causes. Accordingly, the entire background mathematics were completely hidden to the designer, thus eliminating the need for a specific expertise in abstract mathematics and the error-prone re-modeling of the system for mathematical analysis.

Up to now three transformations were elaborated in the HIDE environment. The first one targets formal verification of dependability-related attributes like freedom from deadlocks, avoidance of unsafe system states. A transformation has been defined and implemented which maps a subset of UML statechart diagrams to Kripke structures for formal verification using the model checker SPIN [LMM99a]. The transformation is proved correct with respect to the properties defined in the UML standard. The next transformation targets system-level dependability modeling that covers redundancy structures and fault tolerance schemes. Structural UML diagrams are transformed to Timed Petri Net dependability models [BMM99b]. The analysis helps the designer to identify dependability bottlenecks and to compare different architectural solutions. The third transformation, which is the topic of the current paper, targets detailed quantitative analysis of dynamic behavior.

The HIDE framework thus integrates in a user-friendly way the standard design language UML with a set of validation, verification and evaluation techniques for assuring the quality of service of the system during the early design phases. Design refinement is driven by the information gained during the validation process, thus allowing adequate system designs to be produced before implementation and experimental validation. This allows to shorten the necessary validation cycle.

2.1. The HIDE Environment

The HIDE environment is built up from three main components.

The user-end *modeling platform* is an UML CASE tool, in which the designer can build up his/her UML model. All

[2]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.

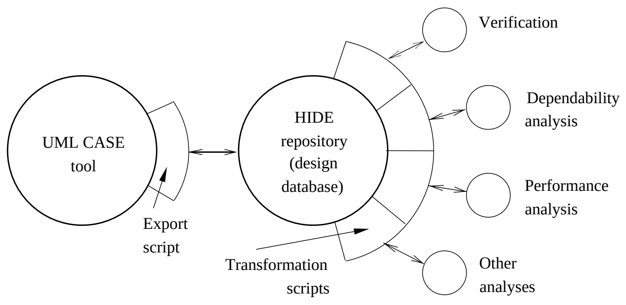

FIGURE 1. The HIDE environment

tool-provided features like code generation, round-trip engineering and documentation generation, can be used without modifications. However, during the creation of the UML model some new constraints and extensions determined by the formal analysis tools must be observed.

The *analysis tools* are off-the-shelf components. The HIDE core acts as an ”end-user” toward the analysis tools by supplying the input for the tool and then by processing and back-annotating the result. The proper modeling formalisms and supporting tools are selected in order to be able to perform the analysis required by the designer.

The *HIDE core* establishes the bridge from the UML CASE tool to the various analysis tools which require different input formats (based on different mathematical formalisms). Instead of implementing a set of direct model transformations from the model repository of the CASE tool to the input formalism of each analysis tool, the HIDE environment incorporates a common representation, the so-called HIDE model repository, which enables to develop the model transformations in a uniform framework. The structure of this database corresponds to the structure of the UML metamodel. Thus, full compliance with the UML standard and a straightforward mapping from the product-dependent repository of the CASE tool is assured. Based on the common representation, specialized scripts (rules) implement the automatic model transformations to the input formalisms of the analysis tools (Figure 1).

Two versions of the transformation framework exist. In the first one, the model transformations were implemented in standard database language (PL/SQL). In the new version, the rules of the transformation are described by a high-level, visual graph language [VVP00] which is interpreted over the HIDE repository.

2.2. Analysis Formalisms and Tools

In our case the quantitative analysis of UML statechart diagrams is performed by transforming them to Stochastic Reward Nets (SRN). The HIDE environment is utilized to define and implement the transformation.

SRN are a GSPN-like formalism based on a semi-Markov reward process [CBC$^+$92, MCT94]. 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 0 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 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 the 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.

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 [CMT89] and PANDA [AD97] were used in our analysis environment (both of them have a compatible input format called CSPL, the C-based SPN Language). PANDA allows 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 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 delivers 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

Guarded Statecharts (GSC) are a sub-class of UML statecharts. GSC 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 Guarded Statechart are states (container states, basic states, and initial states) and transitions with guards. Labels of transitions describe timing information, e.g. arrival distribution of signals, or static information, e.g. 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 also non-deterministic behavior can 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.

We restrict guards of a transition by stipulating that, if a guard contains more than one state of *S*, the predicates of these states are OR-connected. The transition is executed atomically and instantaneously, if its trigger and its guard evaluate to TRUE. The execution effects the nondeterministic choice of exactly one state of the set of target states as next state of the GSC. A guard expression of a GSC *M* may not contain predicates of states of *M*, it may, however, contain state predicates of a concurrent GSC. If such a guard evaluates to TRUE, *M* takes one of the target states irrespectively of its actual state.

An example of a transition is the following:
`startsignal_on [in(M.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* 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 GSC

In this section we indicate how GSC can be use to model the behavior, for example, of an embedded control system.

Using GSCs we can abstract continuous signals to discrete signals assuming a finite set of critical values. For example, it is only important to observe whether a robot arm is directed in a position allowing for unloading, or pointing toward 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. Analogous, 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 [Dou98].

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. Produce the component models. Specific states (the public states) describe the events, system components (controllers and controlled units) generate or respond to. These states represent, for example, sensor and actuator signals. The controllers manage disjoint sets of actuator signals. The modeling of controlled units, usually, needs not to 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 faults, e.g. 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 state transition rates and branching probabilities (weights). Transition rates label timed transitions and specify the mean transition time. Weights label immediate, timeless transitions. They can specify alternatives.
4. Specify the performance and dependability measures. These measures can be expressed in terms of reward functions [CBC^{+}92], assigned to the UML model in the form of structured comments.

3.3. From GSC to Stochastic Reward Nets

For a performance and 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 place holds the name of the basic state. The initial marking of the 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 such 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 Guarded Statechart 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 [OMG97] and on the formalization presented in [LMM99b].

In the next subsection the transformation from UML statecharts to SRN is discussed. Our transformation is presented in a modular way, by introducing a set of SRN *transformation 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 [OMG97].

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.

4.1. Semantics of models

While checking 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 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 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 configuration and the possible transitions triggered by the selected event.

The dynamic 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. 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 (the order is not specified).

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 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 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 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.

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 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. The consistent evaluation of the guards has to be forced in the SRN.

The main distinguishing feature of the semantics of UML statecharts and of 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 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.

4.2. Transformation patterns

The general transformation patterns introduced above are presented in [HM00]. In this section we show typical applications of the patterns by subnets (corresponding to the example in Section 7).

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 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.

4.2.1. Event queue and dispatcher

We have defined two patterns for event dispatchers [HM00]. 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). These dispatching policies are adequate for different applications. Both of them can be extended also to support multi-level priority dispatching.

Figure 2 shows a subnet corresponding to the pattern for nondeterministic event dispatcher. Tokens representing the events *ask*, *down* and *up* are collected in places *ask0*, *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 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 side. For example if an *up* event triggers two concurrent UML transitions then the SRN transition *split_up* has to be inserted to generate tokens in two places *uncons_0* and *uncons_1*.

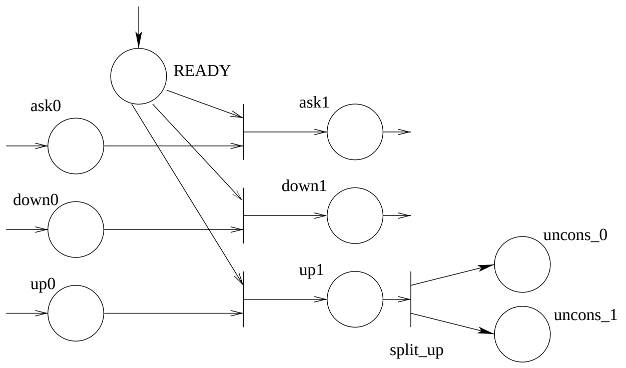

FIGURE 2. SRN pattern of a non-deterministic event dispatcher

Figure 3 shows a subnet belonging to the pattern for the FIFO event dispatcher. The pattern presented here depicts only two kind of events (*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.

There are three columns (of the length of the FIFO) of places: the left-most group is controlling the FIFO structure, the other two groups are for storing the different events. The tokens representing the incoming events arrive at the top of the figure to places *up0* and *down0*, and the just selected one is issued at the bottom in place *up1* or *down1*. 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.

If the queue is full, the incoming tokens will be discarded (by transitions *discard_up* and *discard_down*), else they are placed in the uppermost place of the column corresponding to the type of the event (*up_queue_2* or *down_queue_2*, respectively). Simultaneously a token is generated in the uppermost place (*queue_2*) 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 (tokens

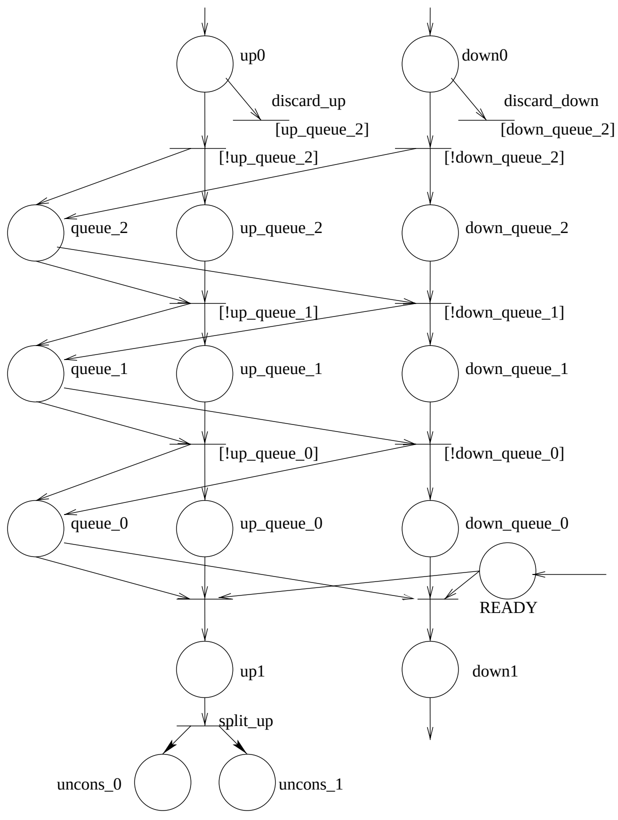

FIGURE 3. SRN pattern of a FIFO event dispatcher

are forwarded by the "split"-transitions to the places representing unconsumed events).

4.2.2. Hierarchy of states and 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), or an AND-state (containing only concurrent sub-machines).

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) may fire at the same time in a single step. 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 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.

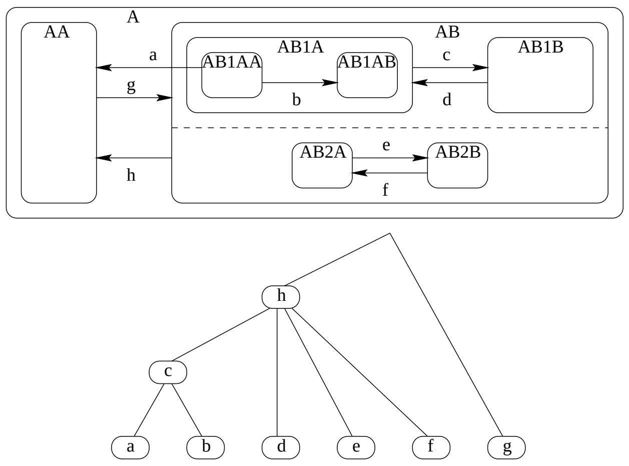

FIGURE 4. The tree structure of the priority relation

Figure 4 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 at the bottom of the figure. The structure of the tree strongly depends on the priority structure of the transitions to be transformed.

The tree structure can be considered as a tree-like daisy-chain of the UML 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 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 subnets (each of them representing a single UML transition) with an auxiliary control structure. This control 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 5(b) shows the SRN pattern of a simple (i.e. not joining) node of the tree, belonging to the UML transition presented in Figure 5(a). The UML transition is represented by the SRN transition *up_t1*. The places of the SRN subnet represent the following items:

- Predecessor states, i.e. states to be left when the UML

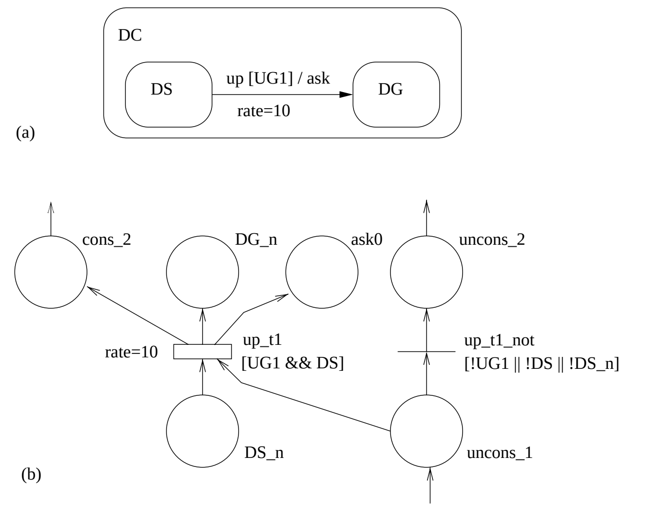

FIGURE 5. A simple UML transition (a) and the corresponding SRN pattern (b)

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 parent states which are not parent states of the target state. They can be identified by looking at the static structure of the statechart. Here both the predecessor and target state are substates of the common state *DC*.

There could be also other states to be left, namely the active states of parallel regions of the SC. These states can not be identified unambiguously by the static analysis of the SC, thus exiting these states necessitates an other construction (described later). These states would not be represented in the SRN corresponding to the transition.

- Successor states, i.e. 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 several arcs of the appropriate tree structure of the triggered transitions. In Figure 5, places *uncons_1* and *uncons_2* 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_2*), if the transition *up_t1* fires. If *up_t1* cannot fire, *up_t1_not* fires, putting the token to place *uncons_2*, i.e. the event remains unconsumed. (The guard of the transition *up_t1_not* describes that *up_t1* cannot fire.)
- Event sending by the transitions is implemented by outarc(s) from the timed SRN transition to the appropriate place(s) of the event dispatcher Here a token is passed to the place *ask0*.

The guard of *up_t1* contains the expression *UG1* (belonging to the guard of the UML transition) and *DS* that refers to the predecessor 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 details in section 4.2.4. 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 a simple timing policy was chosen, where the fastest of the enabled conflicting transitions can fire. There are other possible policies as well, some of them are described in section 4.2.3.

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 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. Two conflicting transitions cannot fire in the same step, because the one of them firing 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.

A joining node of the tree only merges the event chains of the subtrees (Figure 6). 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.

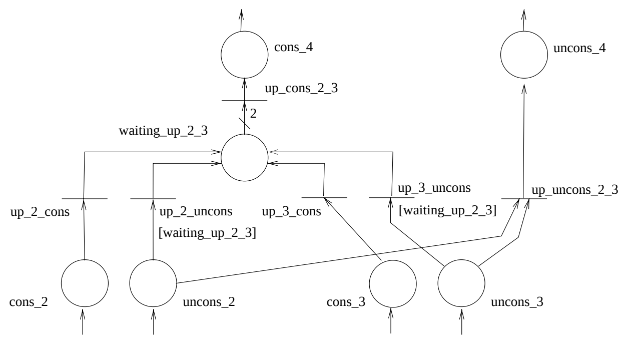

FIGURE 6. SRN pattern of a joining node in the tree structure

The event is already "consumed" in 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 step have to be finished.

In our example the two joining arcs are represented by the place pairs *cons_2*, *uncons_2* and *cons_3*, *uncons_3*. According to the previous pattern (Figure 5), one token can be found either in place *cons_2* or in place *uncons_2*, and another token either in place *cons_3* or in place *uncons_3*.

If the event was not consumed by the transitions on the joining arcs, then there are tokens in places *uncons_2* and *uncons_3*. In this case transition *up_uncons_2_3* 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 *uncons_4*) or, if there are no transitions with lower priority, a token is put to the place *READY*.

If the event was consumed by one or both of the transitions on the joining arcs, then there is a token in place *cons_2* or/and in place *cons_3*. Thus, transition *up_2_cons* or/and *up_3_cons* can fire. Token(s) will be put to the place *waiting_up_2_3*, which may enable to remove the token from the place representing an unconsumed event (if any). If there are as many tokens in place *waiting_up_2_3* as the number of arcs to be joined (here 2), then transition *up_cons_2_3* will fire and a token appears in the place *cons4* representing on the common arc that the event was already consumed.

It can be proved that the properties of the UML SC semantics are satisfied by these patterns, i.e. an SRN transition 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.

4.2.3. *Semantics of timed transitions*

The relationship of 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 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.

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 7 shows the implementations):

- 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). 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

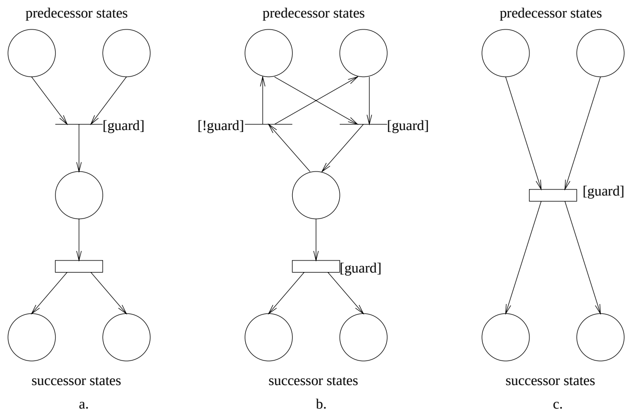

FIGURE 7. Models for combining guards and timing

that the transition can only fire if the appropriate state was active before the actual step. The three figures show 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 SC transitions. 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).

4.2.4. *Step semantics*

The UML semantics requires the evaluation of the guards of the transitions at the beginning of a step, before 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 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_n* 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 *DS_n* and *DG_n* in Figure 5 depict the *next* places, while the guards of the appropriate transitions in the subnet 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. 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*. 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 (we used a depth first order). 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 increase the state space of the SRN without any further advantage. The fixed ordering avoids this kind of state space explosion.

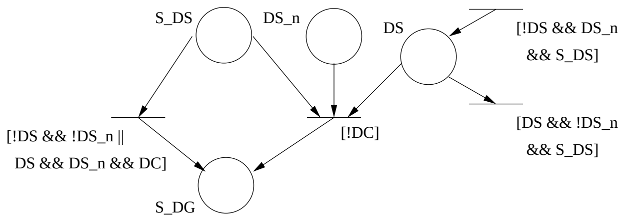

FIGURE 8. Synchronization of the duplicated places

Figure 8 depicts the synchronization pattern of state *DS*, where places *DS* and *DS_n* are synchronized. There is a token in *DS* if and only if the state *DS* of the SC 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 SC will be active after the actual step. The place *DC* represents the direct parent state of *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 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 an SC 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 5 are only the source and parent states of the SC transition, 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. Note that this vanishing problem does not affect the result of the step.

For example, on Figure 4 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.

4.3. Composition of subnets

The SRN corresponding to a given UML statechart is composed of the subnets (transformation patterns) like those presented in the previous sections. 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). Global event dispatching, event dispatching per objects, event dispatching per statecharts, FIFO or non-deterministic dispatching can be selected.
- There are as many transition hierarchy trees as the number of events handled by the transitions of the statecharts of each event dispatchers.
- 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 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. FAULT MODELING

In this section it is shown how faults and errors can be modeled by defining appropriate fault/error models. 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 [LA90].

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. Such state perturbations can be modeled by binary and reflexive relation over the state space of a SC [Dal98a, Dal97, Dal98b, Hus98].

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

Finally, using guards also dependability requirements, expressed as negations of fault trees over component states, can be integrated. This way, dependability requirements, resulting from the requirement analysis, can directly be integrated into the system model. For instance, a fault tree defining possible collisions of certain devices, that could lead to the failure can be specified as guard expression.

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 9). 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(...)* 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'*.

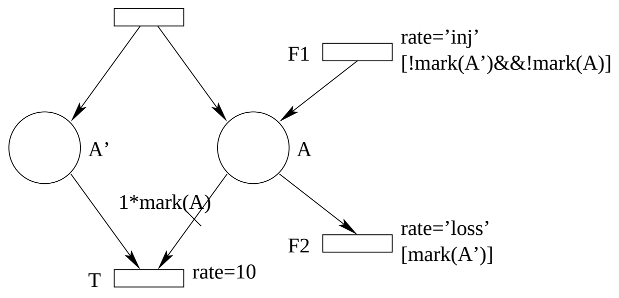

FIGURE 9. Modeling of corrupt signals

There are four cases:

1. Both places are empty, the transition *T* can not fire.
2. Both places contain tokens, the transition *T* can fire.
3. Only *A'* contains a token, i.e. the fault 'signal is lost' has been injected. Then the transition *T* can fire.
4. Only *A* contains a token, i.e. a spurious signal has been injected. Then the transition *T* can not fire. However, the guards of other transitions (which refer to this public state) evaluate to TRUE.

These faults are injected by the transitions *Fl* and *F2*. The modeler has only to provide the corresponding failure (firing) rates *'inj'* and *'loss'*.

6. MODEL ANALYSIS

The model can be analyzed by the SRN tools PANDA or SPNP. In certain cases (in the case of exponential transition firing times) 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 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.

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 repair is modeled). Analogously, safety figures can be derived by distinguishing the unsafe states in the model. Other, application-specific measures may combine performance characteristics with fault modeling (e.g. the performance of the system in the case of an error, utilization of a repair facility, etc.).

The analysis of detailed GSC and statechart models is very time consuming and needs high-performance computers. Full models of realistic applications are usually above the complexity 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 more detail, while the other system components need not be modeled in details. Here the connection with the system-level structural dependability analysis [BMM99a] 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 from the statechart model certain scenarios and to 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 was also elaborated. Performance characteristics like run time, termination probability of selected scenarios can be computed by the SRN tools [DHK99a].

7. AN EXAMPLE

Although the mentioned transformation procedures have been worked out, no examples of applying the transformations for large systems are available yet. Accordingly, we are not able to provide real quantitative assessment of the transformation, however, some qualitative remarks are possible.

We illustrate our approach by a small example of a fault-tolerant system, a variation of a production cell model [LT94, MPW97]. 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. Then the repairman (worker) can repair the press. Also the robot arm may stuck and then be repaired by the repairman.

The complete UML model of an extended version of this example is given in [CDH$^+$98]. It comprises a requirement model, an object model, a deployment model and packages.

- The requirement model describes the actors and use cases of the modeled system. Typical scenarios are modeled by sequence diagrams.
- The static view of the system is captured in class, object and deployment diagrams. The object model of the production cell is organized around the four object diagrams: ProductionCell, Controllers, Machines, and Environment. The deployment diagram describes a possible architecture of the system and shows a given assignment of the components to the nodes; e.g. centralized or distributed control.
- The dynamic view of the system is given by the 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 following, we concentrate on the dynamic view in the form of GSC and full statechart diagrams.

The complete GSC model comprises 5 statecharts (with 9 state transition diagrams and 34 basic states, of which 8 are sensor states and 8 are actuator states). The GSC model of the press (Figure 10) consists of two components, one for the hardware of the press, and one for its controlling unit. This part of the model contains 2 sensor and 4 actuator states. (The guards of some transitions on the figure apply to states of another 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 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. To illustrate the modeling, the statechart corresponding to the hardware of the rotary table is depicted on Figure 11.

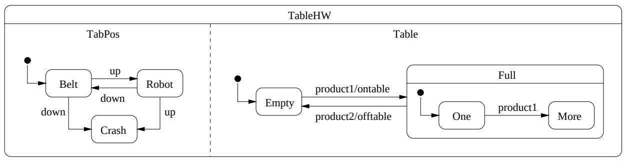

FIGURE 11. Statechart model of the table (hardware)

For the quantitative analysis of the models, the SRN tool PANDA was used. The transformed GSC model (as the components are strongly coupled by the guards) has 9316 reachable states. The size of the state space of the full statechart model increases if FIFO dispatching policy is selected; the increase depends heavily on the length of the queue.

With PANDA, for example, the following parameters can be examined: absorbing states of the system or of its components, the number of reachable states of the system, the expected number of firings of a given transition until an given point in time, the expected time the system spends in a given state until a given point in time. From these data performance and dependability measures (defined by reward functions) like throughput, utilization, mean turn-around time, reliability, availability, etc. can be derived.

Various performance and dependability results were computed [DHK99a]. For example, computing the utilization of the repairman as 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 function of the signal loss rate. There is a domain between 10 and 1000 where the throughput is particularly sensitive to the loss rate (the throughput rapidly decreases to 20%).

Special scenarios like the break-down 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 in average 64s 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 presented a method which allows quantitative dependability and performance analysis of systems modeled by using UML statechart diagrams. To find a 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 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

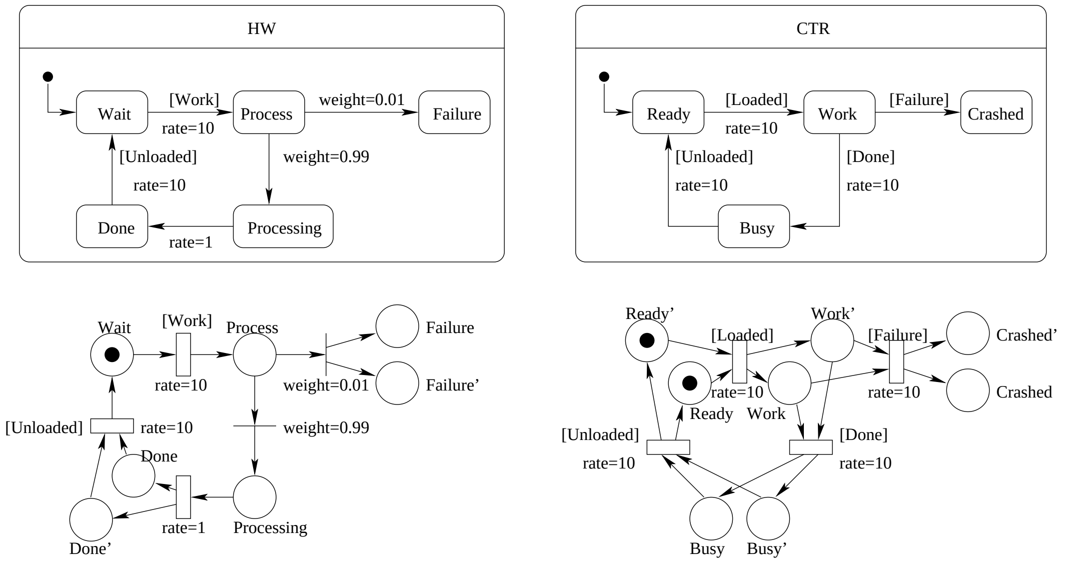

FIGURE 10. GSC model and the corresponding SRN model of the press

be computed. This way the possibility of UML to model and analyze error-prone and fault-tolerant system behavior is greatly enhanced. 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 transformations were presented in the form of transformation 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.

ACKNOWLEDGEMENTS

This work was partially supported by the projects ESPRIT Open LTR 27439 ‘HIDE’, the Hungarian-German Researchers Exchange Program No. 8., OTKA-F030553 and OTKA-T30804 (Hungarian NSF).

REFERENCES

AD97: S. Allmaier and S. Dalibor. 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, 1997.

AM91: 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.

BBK94: 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.

BDLP99: 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.

BG96: 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.

BMM99a: A. Bondavalli, I. Majzik, and I. Mura. Automated dependability analysis of UML designs. In *Proc. 2nd IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’99)*, Saint Malo, France, 1999.

BMM99b: 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.

CBC$^+$92: 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.

CDH$^+$98: Gy. Csertán, M. Dal Cin, G. Huszerl, J. Jávorszky, K. Kosmidis, A. Pataricza, and Cs. Szász. The demonstrator. Technical report, Project deliverable HIDE/D5/TUB/1/v2, 1998.

CMT89: G. Ciardo, J. Muppala, and K. S. Trivedi. SPNP - stochastic Petri net package. In *Proc. IEEE 3rd Int. Workshop on Petri Nets and Performance Models (PNPM’89)*, pages 142–151., Kyoto, Japan, 1989.

Dal97: M. Dal Cin. Verifying fault-tolerant behavior of state machines. In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop HASE 97*, pages 94–99, Bethesda, Maryland, 1997.

Dal98a: M. Dal Cin. Checking modification tolerance. In *Proceedings of the Third IEEE Intemational High-Assurance Systems Engineering Symposium, HASE 98*, pages 9–12, 1998.

Dal98b: M. Dal Cin. Modeling fault-tolerant system behavior. *Advances in Computing Science*, 3, 1998.

DHK99a: 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.

DHK99b: M. Dal Cin, G. Huszerl, and K. Kosmidis. Transformation of guarded statecharts for quantitative evaluation of embedded systems. In *Proc. EWDC-10 lOth European Workshop on Dependable Computing*, pages 143–148, Vienna, Österreichische Computer Gesellschaft, 1999.

DHR95: 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.

Dou98: B.P. Douglass. *Real-Time UML*. Adddison-Wesley, 1998.

Har87: D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

HM00: G. Huszerl and I. Majzik. Quantitative analysis of dependability critical systems based on UML statechart models. In *Proc. HASE 2000, Fifth IEEE Int. Symposium on High Assurance Systems Engineering*, Albuquerque, NM, USA, November 15-17. 2000.

Hus98: G. Huszerl. Formal verification of fault-tolerant systems. a relational approach to model checking. Master's thesis, TU Budapest/Univ. of Erlangen-Nuremberg, 1998.

LA90: P.A. Lee and T. Anderson. *Fault Tolerance, Principles and Practice*. Springer Verlag, Wien, New York, 1990.

LMM99a: D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

LMM99b: 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.

LT94: C. Lewerentz and Th. Lindner, editors. *Formal Development of Reactive Systems*, volume 891. of *Lecture Notes in Computer Science*. Springer, 1994.

MCT94: 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.

MPW97: G. Matos, J. Purtilo, and E. White. Automated computation of decomposable synchronization conditions. In *Proc. Second IEEE High-Assurance Systems Engineering Symposium HASE 97*, pages 72–77, Bethesda, Maryland, 1997.

OMG97: OMG. *UML Semantics, version 1.1*. Object Management Group, September 1997.

RJB99: J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

VVP00: D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. In *Proc. GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 14 – 21, Technical University of Berlin, 2000.