Quantitative Evaluation of Dependability Critical Systems Based on Guarded Statechart Models

M. Dal Cin, G. Huszerl and K. Kosmidis

IMMD 3 University of Erlangen−Nuremberg
and
Technical University of Budapest

{dalcin,kosmidis}@informatik.uni−erlangen.de, huszerl@mit.bme.hu

Abstract

The paper introduces a method to model embedded dependability−critical systems as AND−composition of Guarded Statecharts which are special UML−statecharts. With Guarded Statecharts we can model the reactive behavior of embedded systems so that their quantitative analysis can be performed. First, we present our motivation for using Guarded Statecharts to express the interaction between hardware and software compo−nents of embedded systems, and to model faults and errors as state perturbations. Then we discuss how these models are transformed into Stochastic Reward Nets amenable to a quantitative dependability analysis. Finally, our approach is illustrated by an example.

1. Introduction

A central 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 dependability 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, it is not only necessary to model the system's behavior; e.g. the embedded control algorithm. Also the system's interac−tion with its environment via sensors and actuators has to be modeled (closed−loop modeling), since the environ−ment can be a source of faults which can give rise to errors in the system's behavior. Thus, dependability evaluation of embedded systems tends to be very com−plex causing the modeling problem to be notoriously elusive and error prone. *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 statecharts comprising so−called Guarded Statecharts (GSC) [6]. Statecharts or state diagrams [10] represent finite state machines and describe the behavior of objects in response to external stimuli, such as sensor signals. Statecharts model reactive, state−driven system behavior. They are, however, not directly amenable to a quantitative analysis. Therefore, a method has to be introduced which transforms a set of concurrent statecharts into a mathematical model that can be evaluated quantitatively. Suitable mathematical models could be a directly generated transition system or a Petri Net. In this paper we present a technique for transforming Guarded Statecharts, consistent with UML semantics, into a set of interacting Stochastic Reward Nets (SRN) [4]. Stochastic Reward Nets are extensions to Generalized Stochastic Petri Nets (GSPN) [1]. GSPNs generalize Petri Nets by assigning a firing rate to each transition.

On the one hand, this gives us the possibility to employ the elaborate and well established Petri Net tools for the quantitative analysis of UML−models. On the other hand, this integrates the use of Petri Nets into the object−oriented modeling paradigm of UML. For example, the generated Petri Nets models can be extended by model−ing aspects, difficult to express directly in UML, like the loss or spurious generation of signals (cf. Section 4).

We proceed as follows. In Section 2, we introduce the notion of Guarded Statecharts. In Section 3 it is shown how faults and errors can be modeled by defining appropriate fault/error models. The transformation to

SRN is presented in Section 4. Our approach is illustrated by an example in Section 5 and some results are given in Section 6. An alternative approach to evaluate GSC−models is discussed in Section 7.

2. Guarded Statecharts

The main objects of a Guarded Statechart are states (container states, basic states, initial states, etc.) and transitions with guards. In addition, labels of transitions describe timing information, e.g. arrival distribution of signals, or static information, e.g. probabilities of possible outcomes.

Guarded Statecharts: Given a set E of *external* event variables, a Guarded Statechart (GSC) is a finite set A of actions and a finite set S of states. Each state and each event has a name. Actions denote state transitions.

Actions are of the form:
<guard>*&&*<trigger>*<set_of_target_states>;

- The trigger is a boolean expression of atomic predicates over events.
- Guard is a boolean expressions of predicates *in(state)* where *in(state)* evaluates to true, if *state* is the (actual) initial state (i−state) of the GSC or of some concurrent GSC.

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

For instance,
in(s5) AND in(RHW_off) OR
in(s6) AND in(RC_loads) AND
in(PRC_unloaded)&&TRUE{s6}*;
State s6 is the target state of this action. The trigger is the constant expression *TRUE*, not mentioned in Figure 1.

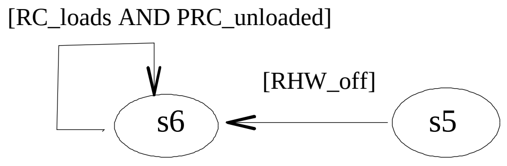

Figure 1 Action

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. For instance, a task can be requested at any time and peripherals may react to controls with unpredictable delays due to faults.

We restrict guards of an action by stipulating that, if a guard contains more than one state of S, the predicates of these states are OR−connected. The action 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 <set_of_target_states> as next i−state of the GSC. A guard expression of a GSC M may not contain predicates of states of M. If such a guard evaluates to *TRUE*, M takes one of the target states irrespectively of its actual i−state. That is, the OR−connection of all state predicates of M always evaluates to *TRUE*.

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

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 (Figure 2). 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 (public states). Likewise, 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 multithreaded, because it models a message being passed to another object without the yielding of control [9].

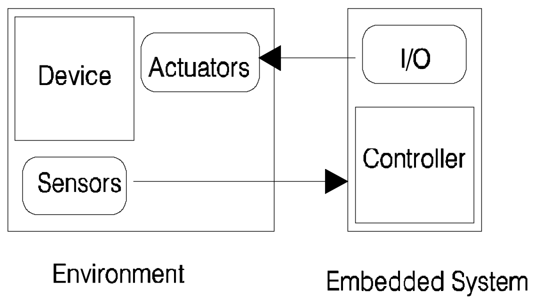

Figure 2 Modeling view

The following steps lead to a 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 (so−called public states) describe the events, system com−ponents (controllers and controlled units) generate or respond to. These states represent, for example, sen−sor and actuator signals. The controllers manage dis−joint sets of actuator signals. The modeling of con−trolled 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 ac−tuator 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, time−less transitions. They can specify alternatives.
4. *Specify the performance and dependability measures.* These measures can be expressed in terms of reward functions [4].

3. Modeling Faults and Errors

Truly dependable systems are able to cope with faults. Following types and locations of a fault can be distinguish. 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 (Figure 3). Faults can give rise to errors, that is to undesired system states, which in turn can lead to the failure of the system [13].

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 for GSCs 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 GSC [5,7,8,12].

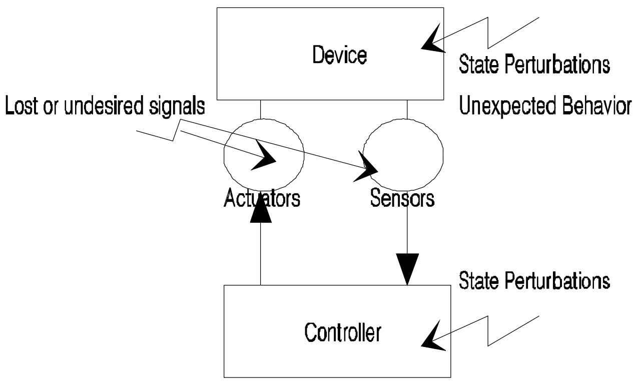

Figure 3 Faults

Signal losses can cause that guards are not observed. For example, the guard *in*(*RHW_off*) may not be observed by the robot control. The guard then always evaluates to *TRUE*. This way, also sensor and actuator faults or loss of messages can easily be modeled by state perturbations.

Finally, using guards also dependability requirements, expressed as negations of fault trees over component states, can be integrated into GSCs. This way, depend−ability requirements, resulting from the requirement analysis, can directly be integrated into the system model. For instance, a fault tree defining possible colli−sions of certain devices, that could lead to the failure can be specified as guard expression, see Figure 4.

4. From Statecharts to Stochastic Reward Nets

For a dependability analysis the GSC−models must be transformed to models amenable to mathematical analysis. Guarded Statechart can easily be transformed into Stochastic Reward Nets (SRN). State transitions with time delay are transformed to timed Petri Net transitions and time−less transitions to immediate Petri Net transitions. Guards and triggers become guards of Petri Net transitions. These SRNs can then be evaluated with a PN/SRN tool.

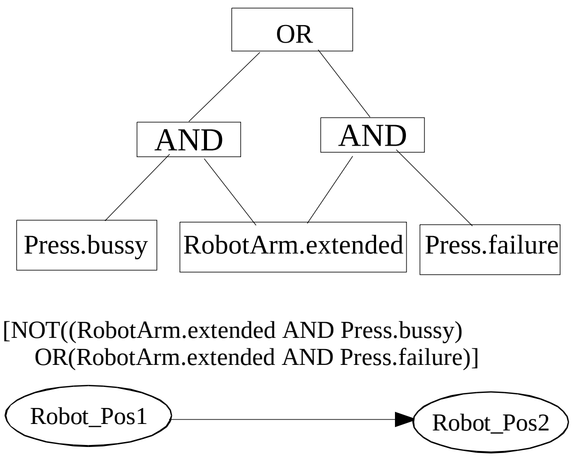
OR
AND
AND
Press.bussy
RobotArm.extended
Press.failure
[NOT((RobotArm.extended AND Press.bussy)
OR(RobotArm.extended AND Press.failure)]
Robot_Pos1
Robot_Pos2

Figure 4 Fault tree and its representation as guard expression

PANDA [2], our Petri Net analysis tool, 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 stringent and clear reward concept has been developed 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). This function delivers the number of tokens in a place. PANDA computes the expectation value of a reward function at a point in time (e.g. availability or throughput) as well as accumulated rewards. PANDA is available for shared and distributed memory platforms.

GSCs are not hierarchic − rather, all hierarchy levels (except the bottom level) describe concurrent behavior. The transformation neglects all these concurrent con−tainer states, since they have no counterparts in the Petri Net structure of an SRN. The basic states are represented as places. The PN−place holds the name of the GSC−basic state. The initial marking of the place is 1, if there is an initial transition in the GSC leading to the corre−sponding state. Otherwise the initial marking is 0. Additional PN−transitions are generated for loss or gen−eration of spurious signals. The modeler has only to specify the rates.

The main transformation steps are:

1. *Private states*, i. e., states which do not appear in guard expressions are transformed to places.

2. *Public states*, e.g. sensor and actuator states, are transformed into a pair of places, see Figure 5. The arc annotation 1xMark(...) determines a state dependent capacity of the arc. For example, if Mark(*State_PUB*) = 0, then firing of the output−transition *S* depends only on the marking of place *State*. The duplication of public states serves mainly to model − within the Petri Nets − communication faults, e.g. lost or spurious sensor or actuator signals. A fault occurs when places *State* and *State_PUB* have different markings (see below).

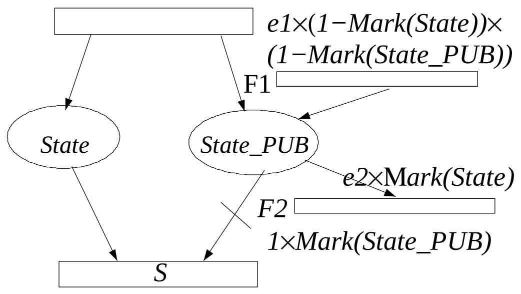

Figure 5 Modeling fault injection

3. *State transitions labeled with rates* are transformed to timed Petri Net transitions with the same rates.

4. *State transitions labeled with weights* are transformed to immediate PN−transitions with the same weight. Immediate transitions have priority over timed transi−tions. The weights of conflicting immediate transi−tions are normalized such that they become branching probabilities. (At present, weighted transitions can not have guards. In fact, they are mainly used to model state perturbations.)

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

As mentioned, our fault model includes corrupted actua−tor and sensor signals. A guard can sense an active signal state as being inactive and vice versa. We duplicate, therefore, the places corresponding to signal states (pub−lic states). Place *State* models the state of the signal and place *State_PUB* models the presence of the signal (Fig−ure 5). There are four cases: 1) Both places are empty, the transition S can not fire. 2) Both places contain tokens, the transition S can fire. 3) Only *State* contains a token, i.e. the fault ’signal is lost’ has been injected.

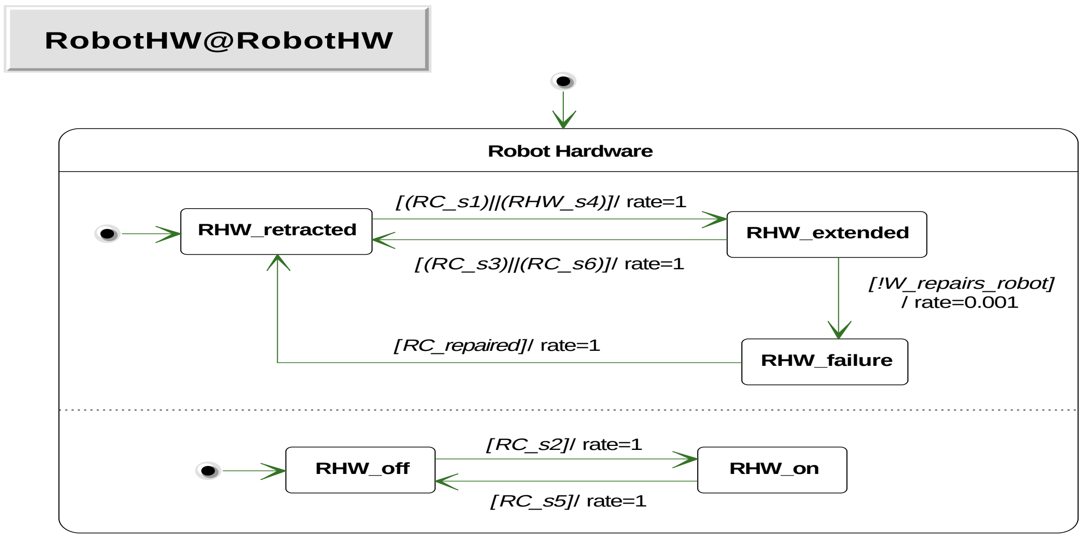

Figure 6 Hardware of robot

Then the transition S can fire. 4) Only *State_PUB* contains a token, i.e. a spurious signal has been injected. Then the transition S can not fire. However, the corresponding guard evaluates to *TRUE*. The faults are injected by the transitions F1 and F2. The modeler has only to provide the corresponding failure (firing) rates e1 and e2.

The transformation and the SRN−analysis tool are part of the project *High−Level Integrated Design Environ−ment* HIDE developed under EU contract LTR27439 [11]. This project aims at an extension of modern CASE−tools by model−based mathematical analysis and validation techniques.

5. An Example

We illustrate our approach by a small example of a fault−tolerant system. The example is a variation of a production cell model [14, 16]. The system contains a press that processes metal blanks, a robot with an extensible arm (with a 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.

According to our modeling approach, each device model consists of a hardware behavioral model and the corre−sponding control charts. The control charts specify either the behavior of a single, central cell controller or that of several distributed device controllers. 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). These statecharts have to be transformed to SRNs for analysis [6]. Figures 6 and 7 show the statecharts of the robot. We use the UML−modeling tool INNOVATOR [17].

The HW−statechart contains 4 sensor states representing the positions of the robot arm and the on/off−state of the magnet. The 5th sensor state represents a perturbation: the failure of the robot arm to retract (Figure 6). The GSC of the robot control (Figure 7) contains 2 state tran−sition diagrams: a diagram specifying the communication with the press control and the repair console, and a diagram specifying the control of the robot arm. The states RC_si are actuator states. Also a loss rate for signal s6 is specified. The robot control repeats sending this signal until it is received by the hardware.

The complete UML−model of an extended version of this example is given in [11]. It comprises a requirement model, an object model, a deployment model and packages.

- The requirement model describes the requirements to the modeled system. It contains use case diagrams and sequence diagrams.

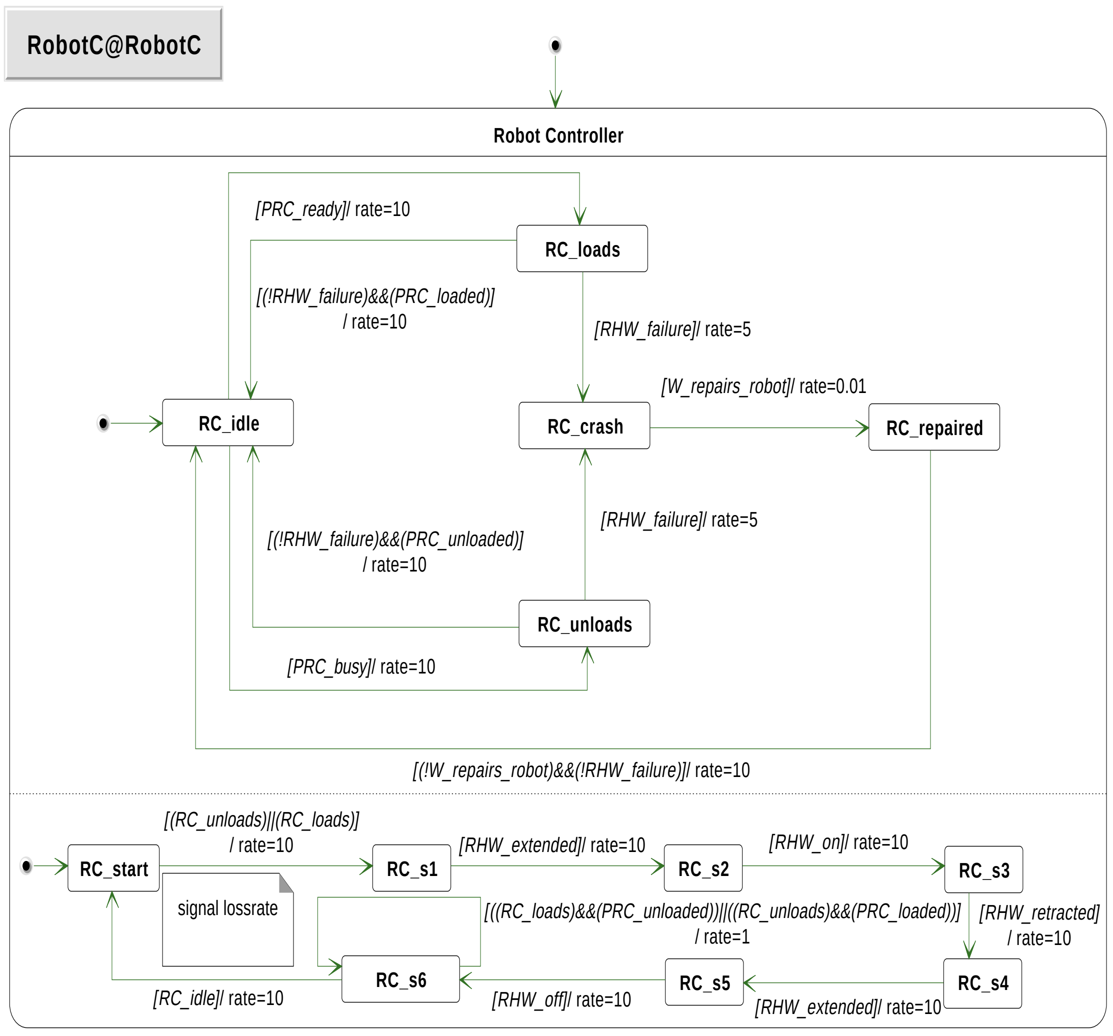

Figure 7 Robot control

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*, *Environment*, and *Machines*.
- The deployment model consists of several deploy−ment diagrams of the system. A 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 package model of the production cell consists of three packages and a package diagram. The *Machines* package includes the nodes of the system that repre−sent the physical machines. The *Controllers* package contains the nodes of the system that represent the physical controllers. The third package contains the components of the system that describe the system's functionality.

The dynamic view of the system is given by the statecharts.

6. Quantitative Results

Some measurements with the transformed models were performed which provided useful experiences [11]. For the quantitative analysis our SRN−tool PANDA was used. The transformed GSC−model of our small example has 63000 states. However, the components are strongly coupled by the guards; 9316 states are reachable from the initial configuration.

With PANDA we can

- detect absorbing states of the system or of its components,

- determine the number of reachable states of the system,
- determine the expected number of firings of a given transition until an given point in time,
- determine the expected time the system spends in a given state until a given point in time.
- etc.

From these data performance and dependability measures (defined by reward functions) like throughput, utiliza−tion, mean turn−around time, reliability, availability, etc. can be derived. Figures 8 and 9 show the utilization of the repairman as function of elapsed time and the throughput of the production cell as function of the signal loss rate. The throughput is the mean number of forged blanks per time unit (1 sec).

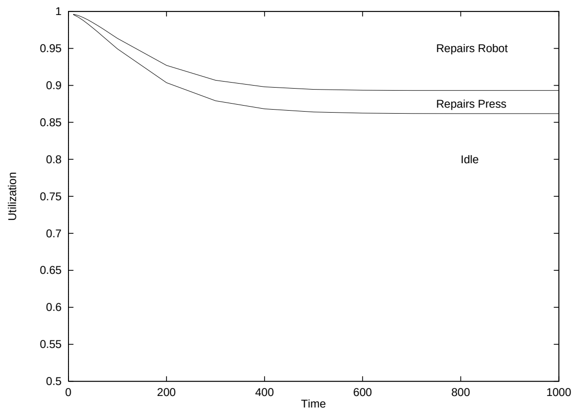

Figure 8 Repairman utilization

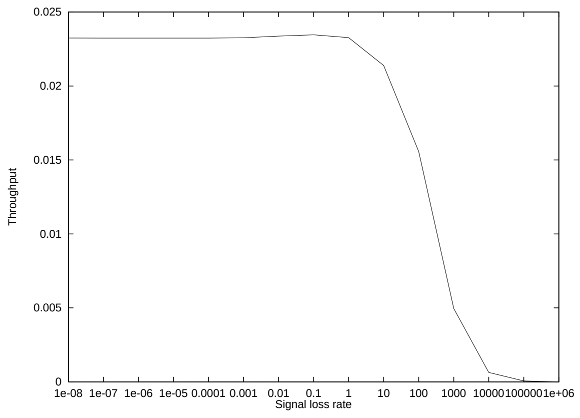

Figure 9 System throughput

7. Scenarios

The analysis of GSC−models needs either high perfor−mance computers or is very time consuming, and even a little more realistic model then that of [11] would cause problems. But in practice the complexity of our (extended) model is near the maximal complexity modern tools can handle. Thus, we have, most probably, to concentrate our quantitative analysis on certain system components such as the embedded controllers. The controllers are modeled in greater details whereas the devices need not be modeled with details. For a dependability analysis it may only be necessary to specify how they develop state perturbations.

Another way to reduce complexity is to deduce from the GSC−model certain scenarios and to model them by se−quence diagrams. Usually these sequence diagrams are much less complex than the GSC−model itself. We transform then the sequence diagrams to SRNs (Figure 10) and, first, check whether the scenario works, e.g. we check whether it is deadlock−free. Then we determine the probability, that the scenario terminates after a given time. For example, the sequence diagram which de−scribes the break−down of the robot arm and its repair is given in Figure 11. The corresponding SRN has only 51 places, 31 transitions, and 15 reachable states.

Figure 13, derived from this scenario, shows the distri−bution function of the time it takes to load the press again after the breakdown of the robot arm. Figure 12 shows the scenario where the signals from the robot control are lost twice and Figure 14 presents the distribution func−tions of the duration of the fault−free scenario (1) and the faulty scenario (2).

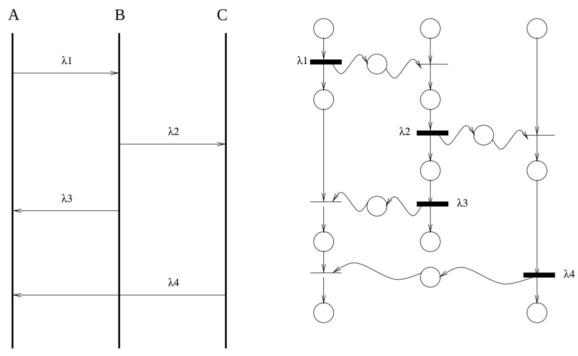

Figure 10 A sequence diagram and its transformation

8. Conclusion

We presented a modeling paradigm for dependability−critical embedded systems and an approach to evaluate the models quantitatively. Our starting point is an object−oriented UML−model of the embedded system. The target are analytical Stochastic Reward Nets amenable to a quantitative analysis. This way the possibility of UML to model and analyze error−prone and fault−tolerant system behavior is greatly enhanced.

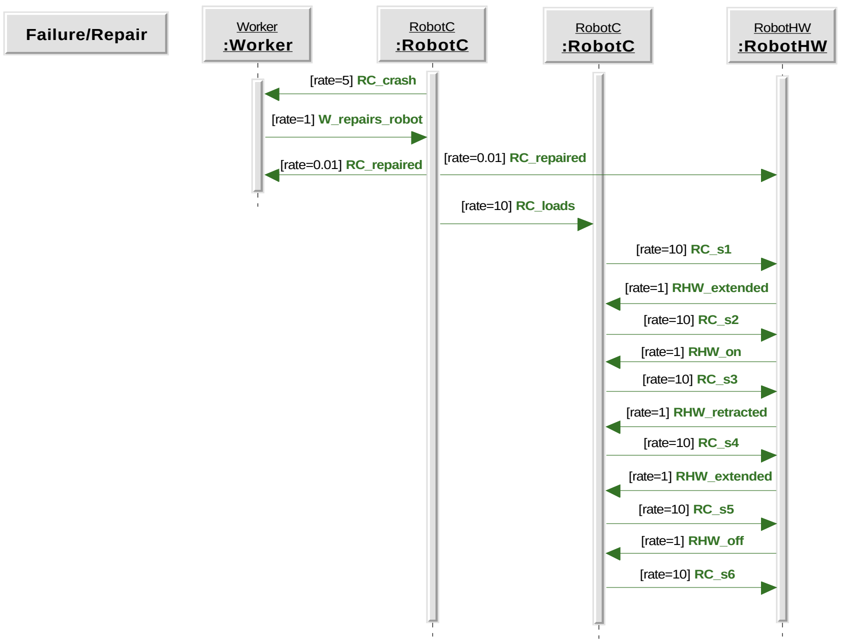

Figure 11 Scenario 1

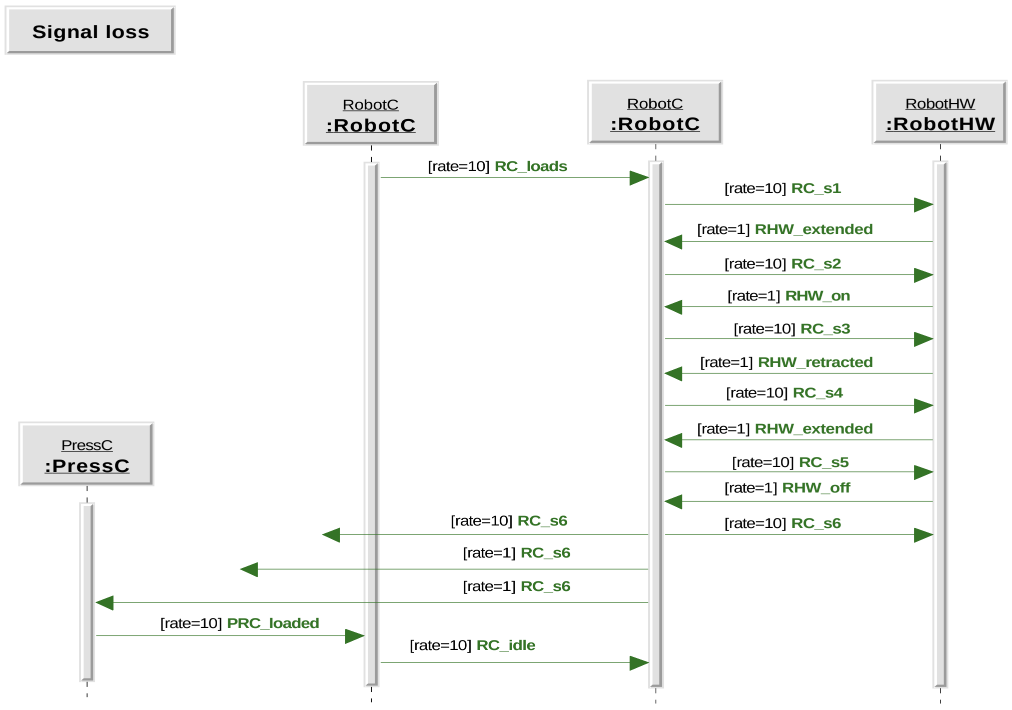

Figure 12 Scenario 2

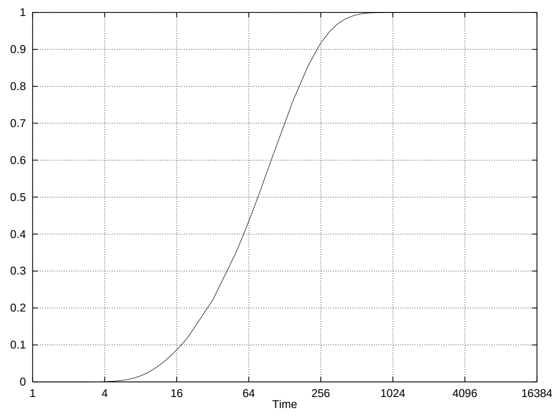

Figure 13 Distribution function for the duration of scenario 1

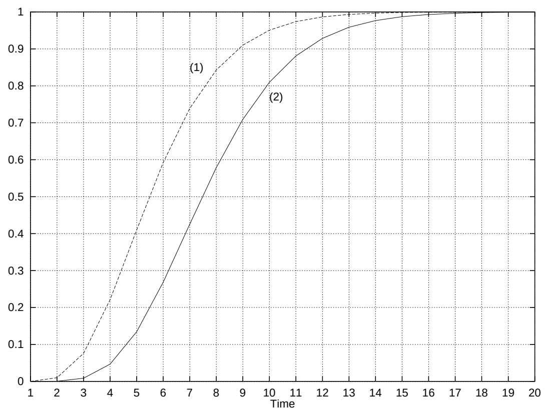

Figure 14 Distribution function for the duration of scenario 2

9. References

[1] M. Ajmone Marsan, G. Balbo, G. Conte (1986): Performance Models of Multiprocessor Systems, The MIT Press.

[2] S. Allmaier, S. Dalibor (1997): PANDA − Petri Net ANalysis and Design Assistant, Tools Descriptions, 9th Int. Conference on Modeling Techniques and Tools for Computer Performance Evaluation, St. Malo, 1997.

[3] S. Allmaier, D. Kreische, Parallel approaches to the numerical transient analysis of stochastic reward nets *ICATPN'99*, Springer LNCS Vol.1639, pp. 147−167, 1999

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

[5] M. Dal Cin: Checking Modification Tolerance, Proceedings of the Third IEEE International High−Assurance Systems Engineering Symposium, HASE 98 pp. 9−12, 1998.

[6] M. Dal Cin, G. Huszerl, K. Kosmidis, Transformation of guarded statecharts for quantitative evaluation of embedded systems, Proc. 10th European Workshop on Dependable Computing. Vienna, Österreichische Computer Gesellschaft, pp. 143−148, 1999

[7] M. Dal Cin: Verifying fault−tolerant behavior of state machines. In Proceedings of the Second IEEE High−Assurance Systems Engineering Workshop HASE 97, Bethesda, Maryland. IEEE: 94−99, 1997.

[8] M. Dal Cin (1998): Modeling fault−tolerant system behavior, in Advances in Computing Science Vol. 3, Springer Verlag Wien New York.

[9] B.P. Douglass (1998): Real−Time UML, Adddison−Wesley.

[10] D. Harel (1987): Statecharts: a visual formalism for complex systems. Science of Computer Programming 8: 231 − 274

[11] HIDE: High−level Integrated Design Environment for Dependability, University of Erlangen−Nuremberg (FAU−IMMD3), Consortio Pisa Ricerche − Pisa Dependable Computing Centre (PDCC), Technical University of Budapest (TUB−MIT), MID GmbH, INTECS Sistemi S.p.A.

[12] G. Huszerl (1998): Formal verification of fault−tolerant systems. A relational approach to model checking. Diploma thesis, IMMD3

[13] P.A. Lee, T. Anderson (1990): Fault Tolerance, Principles and Practice. Springer Verlag, Wien NewYork

[14] C. Lewerentz, Th. Lindner (Eds.) (1994): Formal Development of Reactive Systems, Springer Lecture Notes in Computer Science Vol. 891

[15] P. Liggesmeyer, M. Rothfelder (1998): Towards automated proof of fail−safe behavior, in Computer Safety, Reliability, and Security, Springer LNCS Vol 1516: 169−184

[16] G. Matos, J. Purtilo, E. White (1997): Automated computation of decomposable synchronization conditions. Second IEEE High−Assurance Systems Engineering Workshop HASE 97, Bethesda, Maryland. IEEE: 72−77

[17] MID: INNOVATOR, MID GmbH Nürnberg