Validating Timing Models of Industrial Real-Time Systems

Johan Andersson, Anders Wall, and Christer Norström

Department of Computer Science and Engineering, Mälardalen University,
Box 883, Västerås, Sweden,
{johan.x.andersson,anders.wall,christer.norstrom}@mdh.se

Abstract. When analyzing a model of any kind, in order to get confidence in the analysis result it is necessary to have confidence in the model. If a model describes the timing of a complex software system, it is not obvious how to determine if the model is valid, i.e. if the model accurately describes the system from a certain point of view and at an appropriate level of abstraction. Given that a model is regarded as valid, to facilitate future maintenance of the model it should also be verified that the model is robust with respect to typical changes. In this paper we propose methods for establishing confidence in the validity and robustness of models describing the temporal behavior of software.

1 Introduction

As large industrial software systems evolve, their software architecture may degrade. This since maintenance activities are often performed in a less than optimal manner due to resource restrictions e.g. limited time budgets. As a result of these maintenance activities, not only the size but also the complexity of the system increases. Eventually it becomes hard, or even impossible, to predict the impact that changes will have on the system's behavior. As a consequence of the low understandability of the system's behavior, the engineers are dependent on extensive testing, which is time-consuming and costly. By introducing analyzability with respect to properties of interest, the understandability of the system can be increased.

If the software system has real-time requirements, it is of vital importance that the system is analyzable with respect to timing related properties, e.g. deadlines. Introducing analyzability, and consequently introducing the possibility of understanding the impact that changes will have on the system behavior with respect to timing, can be done in two distinct ways: *intrusively* or *non-intrusively*. In an intrusive approach the system is re-designed in order to make it analyzable. An example of an intrusive approach is switching from event triggered scheduling to time triggered scheduling or introducing a server algorithm to handle aperiodic tasks. The intrusive approach is, however, associated with a high cost as it might require a considerable effort to change the system. It is also a risk since errors might be introduced that, in worst case, is not captured during testing.

In a non-intrusive approach a model of the system is constructed. Hence, the system is kept intact and unchanged which minimizes the cost and the risks. The work presented in this paper focuses on a non-intrusive approach which has been developed as part of a case study. A probabilistic modeling and analysis framework was developed and a statistical model was constructed describing the temporal behavior of the ABB Robotics robot control system, which is a complex industrial real-time system [8]. While constructing the model we discovered that it was not clear as how to validate the model.

The validation of a software model is the process of determining whether or not the model is a correct description of the system with respect to the properties of the system that the model is intended to describe. This is typically done by comparing observations of the system's behavior with the predictions made by analyzing the model. Moreover, in order to facilitate future usage of the model, it should be easy to keep the model and the system consistent as the system evolves. The effort of adjusting the model to reflect the impact of a maintenance operation should not be similar to constructing the initial model, the change required to update the model should be intuitive and similar to the change in the system. Therefore, it is necessary to verify that the model is *robust* with respect to typical types of changes of the system.

In this paper we propose a methodology for validation of models describing the temporal behavior of complex real-time systems. We define an equivalence relation between a model and the corresponding system. The equivalence relation is used for assessing the validity and the robustness of such models.

The outline of this paper is as follows: Section 2 describes related work, in Section 3 we outline potential error sources when constructing a model, in Section 4 we discuss how to compare a timing model with the temporal behavior of the real system and define an equivalence relation between timing models and system implementations, in Section 5 we propose a method for analyzing the robustness of timing models by sensitivity analysis. Finally we conclude the paper and give hints on future work in Section 6.

2 Related Work

Validity of models has been studied in the simulation community. In [3], model validation is defined as "the process of determining whether a simulation model is an accurate representation of the system, for the particular objectives of the study". They address validity of models that are to be used for general simulation-based analysis, e.g. simulation of a physical process, but they do not discuss the problems of performing the actual validation when the model describes the timing of a complex software system.

A process for constructing simulation models is described in [1], where the assessment of model accuracy is integrated. The different activities required for quality assurance is described. This process is quite complex as it contains 10 processes and 13 credibility assessment stages. However, this is guidelines on

a quite high level of abstraction. The work does not address what or how to observe and compare the system with a model when validating.

Model validity from a general simulation point of view is also discussed in [4]. Different processes for validation of models are described in the paper, one process is *Independent Verification and Validation*, IV&V. It states that a third party reviewer should be used to increase the confidence in the model. A scoring model is also described, where various aspects are weighted and a total score can be calculated as a measure of validity for the model. This is, as pointed out in the paper, dangerous since it seems more objective than it really is and might cause over-confidence in the model. The author describes a simplified version of the modeling process described in [1], consisting of the Problem Entity (the system), a Conceptual Model (the understanding of the system), and a Computerized Model (the implementation of the Conceptual Model). Furthermore, Conceptual Model validity is defined as the relation between the Problem Entity and the Conceptual Model, i.e. if the person constructing the model has a correct understanding of the system. Operational Validity is the relation between the Computerized Model and the Problem Entity, i.e. if the Computerized model was correctly implemented.

In [3] many aspects of the validity of models in general is discussed and a seven-step approach for conducting a successful simulation study is described. This approach is on a quite high level of abstraction and can be applied on any model. The steps are problem formulation, collecting data and construction of the conceptual model, validation of the conceptual model, programming the model, validation of programmed model, experiments and analysis, and presentation of results. The paper stresses the importance of a definite problem formulation, comparisons between the model and the system, and the use of sensitivity analysis. This is in line with the earlier work of this project [7][8]. This work does not address models of software systems and the difficulties of validating them.

3 Sources of Error in a Model

The need for model validation emerges from the risk of constructing a model that contains errors or lacks information about important details of the system's behavior. The process of constructing a model of a software system consists of several different activities and errors could be introduced in any of them. There are at least four potential error sources:

- the understanding of the system,
- the understanding of modeling language and tools,
- the observations of the system, and
- the level of abstraction in the model.

The understanding of the system The modeling team must understand both the structure and the behavior of the system in order to develop a valid and

robust model. They should discuss their understanding of the system with system experts and let them review the resulting model in order to avoid errors in the conceptual model. This is in line with IV&V [4].

The understanding of modeling language and tools The modeling team must have adequate knowledge about the different tools that are used for modeling and analyses and the semantics of the modeling language. Otherwise, there is an obvious risk of misunderstandings or misinterpretations. To avoid such errors, the tools and modeling language must be well documented and communicated.

The observations of the system When constructing a model based on observations of a systems behavior, it is important that the observations are made in several different but representative situations. This in order to ensure that as much as possible of the behavior of the system is captured. For instance, it is likely that a system that gets exposed to stimuli from its intended environment behaves differently from a system that is in its idle mode. This is further discussed in Section 4.3. Moreover, if software probes are used when measuring the system, the probe effect has to be considered [5]. This is especially important for real-time systems where probes affect the temporal behavior and potentially cause or prevent exceptional events in the system, for instance, a missed deadline or a buffer underrun. One solution to avoid the probe effect is to use specialized hardware that monitors the system without affecting the temporal behavior of the system [6]. Another solution is to leave the probes in the system, so that the impact imposed by the probes is not removed. In this work it is assumed that the probes do not have to be removed but can be left in the system.

The level of abstraction If information about important details of the systems behavior is missing, the model will be less accurate and less robust. A sensitivity analysis, described in section 5, can evaluate whether or not this is the case.

4 Model equivalence

In this section we will present our notion of *equivalence*. The proposed equivalence relation enables a comparison between the temporal behavior predicted when analyzing a model and the temporal behavior observed when executing the system. Since models are abstractions of the system, the predicted behavior will consequently be an abstraction of the behavior of the system. Hence, it is not feasible to compare the predicted behavior with the observed behavior directly.

As an example consider the measured response times and the predicted response times for a task shown in Figure 1. Each dot represents an instance of the task, where the Y axis is the response time and the X axis is the time when the instance started. One instance is one execution of the task. The data presented in this figure was collected in the ABB Robotics case study [7][8] mentioned in the introduction.

We can see that the temporal behavior is mimicked by the analysis although it is rougher. Distinct classes of response times can be identified in the observed

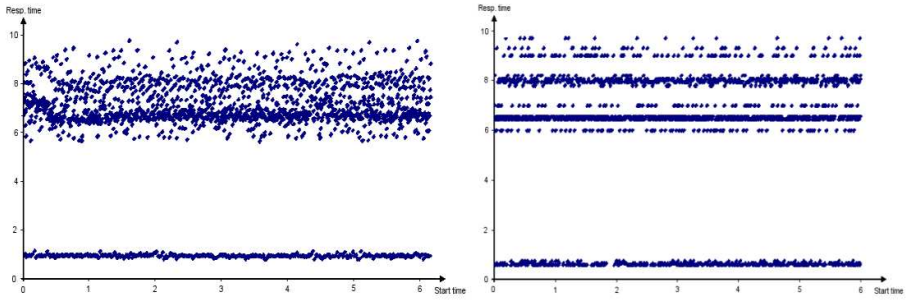

Fig. 1. Observed (left) and predicted (right) response times for a task

and the predicted behavior and these match very well. As mentioned earlier, it is not possible to compare these two data sets instance by instance, as they do not match directly.

Instead, if the system and the model are equal with respect to a set of *system properties* that characterize the temporal behavior of the system, we will say that the model is *observable equivalent* to the implementation. As an example, if the deadline of the task shown in Figure 1 is 10 time units then the model and real system has equal behavior with respect to the property of meeting that deadline.

A system property in our framework is a probabilistic statement regarding an aspect of the behavior of the system, that we can observe directly or derive from observations of a system, but not find explicitly in the implementation or configuration of the system.

Thus, the priority of tasks, the execution time of a task and the rate of periodic tasks are not system properties. The priority and rate of tasks can be found in the implementation and the execution times of tasks can be calculated using tools. Typical examples of system properties are if a tasks response time is less than a specific deadline, if precedence restrictions are violated or the probability of a message queue being empty or being full.

4.1 Using system properties for comparison

When a model is to be compared with its corresponding system in order to establish equivalence, a set of system properties has to be selected as a point of view for the comparison. This set of system properties, the *comparison properties*, are evaluated both with respect to the observed system behavior and with respect to the results from analyzing the model. The comparison properties typically include explicitly defined system properties of interest but may also include other system properties, in order to increase the coverage of the comparison. These *supporting properties* are of low interest when analyzing the model, but contain a lot of information about the behavior of the system. A typical supporting property could be the average number of messages in a message queue.

Selecting the appropriate comparison properties is very important in order to get a valid comparison. As many system properties as practically possible should be included in the set of comparison properties in order to get high confidence in the comparison. If too few relevant system properties are included a partially mismatching model might be accepted and regarded as valid. In [4], this is mentioned as the *model user's risk*. However, if some of the comparison properties are irrelevant, there is a risk of rejecting a valid model due to differences in irrelevant system properties. Rejection of a valid model has mentioned in [4] as the *model builder's risk*.

The selected system properties should not only be relevant, but also be of different types in order to compare a variety of aspects of the temporal behavior. In this paper we have identified four types of timing related properties:

- response-time properties,
- event pattern properties,
- synchronization properties, and
- message buffer properties.

Response-time properties The response time of tasks can be used as a comparison property, since it is dependant on not only the execution time of the task, but also depends on the temporal behavior of other tasks. The response time can be interesting in terms of worst case, since it might be a requirement (a deadline), but also the distribution of response times can be used as a supporting property, as it contains a lot of information about the temporal behavior of the system.

Event pattern properties It is often possible to identify patterns in the execution of tasks and arrival of events. For instance, a system property of this type is the probability of a Task A preempting or preceding Task B. Another system property of this type is the distribution of interarrival times of an aperiodic event. For instance, a property could state that 95 % of the observed events of this type arrived between 7-8 ms after the arrival of the last event of that type. The occurrence of a certain pattern in the execution times of tasks is also a system property that can be used for comparison.

Synchronization properties This type of properties are related to semaphores and their effects, for instance which tasks that blocks other tasks and for how long. Properties of this type could also state the absence of deadlocks and timeouts.

Message buffer properties This type of properties include those related to message buffers, for instance the minimum or maximum number of messages, how long a task waits for a message, how often a task writes or reads messages from the buffer. Another example of such a property is the probability of a certain message buffer being empty or full.

Even if a large set of comparison properties are used, if they represent too few types of system properties, there is a risk of accepting an invalid model, For instance, imagine that only response-time properties are used as comparison

properties. The rate of a task could in that case differ between the system and model without being discovered in the comparison. If comparison properties related to event patterns had been used as well, this would always have been discovered.

4.2 Observed equivalence

As mentioned, in order to determine equivalence between an implementation and a model we specify a set of system properties to be used for the comparison. This set, the comparison properties, contains explicitly defined system properties of interest and supporting properties, as discussed in Section 4.1. We formally define the comparison properties, P_s, as:

Definition 1 *$P_s = \{p_1, ..., p_n\}$, where $n \in N$, is the set of system properties.* □

Since we decide on equivalence based on a comparison between observed temporal behavior and the results from analyzing the corresponding model, we say that it is an *observable equivalence*. We will refer to the observed temporal behavior of the system as a function *Obs*, and the prediction behavior as a function *Pred*. Formally, we define *Obs* and *Pred* as:

Definition 2 *Obs(T_S, Q_S, L_S) = $\langle Start, Exec, Response, Queue, Sem\rangle$ is a function that returns the result from monitoring the execution of a system S, where T_S are the tasks in S, Q_S are the message queues in S and L_S are the semaphores in S. Start is a set containing the start times of all observed instances of the tasks in T_S. The sets Exec and Response contain the execution times and response times of the observed task instances. Queue is the observed number of messages in each of the message queues in Q_M over time, and Sem is the observed status of each of the semaphores in L_M over time.* □

Definition 3 *Pred(T_M, Q_M, L_M) = $\langle Start, Exec, Response, Queue, Sem\rangle$ is a function that returns a prediction of the behavior based on a model M, where T_M are the tasks in M, Q_M are the message queues in M and L_M are the semaphores in M. Start is a set containing the start times of all predicted instances of the tasks in T_M. The sets Exec and Response contain the execution times and response times of the predicted task instances. Queue is the predicted number of messages in each of the message queues in Q_M over time, and Sem is the predicted status of each of the semaphores in L_M over time.* □

There might be tasks in the system S which are not present in the model M ($T_M \subseteq T_S$). Moreover, there might be message queues ($Q_M \subseteq Q_S$) in S which are not in the model as well as semaphores ($L_M \subseteq L_S$). This corresponds to the abstractions made in the model. For more information on modeling and abstraction in our framework we refer to [7].

As mentioned in Section 4.1, system properties are probabilistic statements regarding a systems temporal behavior. For instance, a system property stating

that the response time of task τ must be less than 10 time units with a probability of 1, i.e. a hard deadline, could be expressed as:

$$Probability(\tau.response < 10) = 1$$

The function *Probability* calculates the probability of the condition being true by dividing the number of elements matching the condition with the total number of elements in the set $\tau.response$. Note that the $<$ operator is used to compare a set with a single value. It compares all values in the set specified as left operand with the value specified as right operand, in this case 10, and returns the subset of values are less than the right operand.

This notation for describing system properties is further described in [8]. We will denote the result from applying a system property p on the observed temporal behavior of system S as:

$$p(Obs(T_S, Q_S, L_S))$$

and we will denote the result from applying a property p_i on the predictions based on the model M as:

$$p(Pred(T_M, Q_M, L_M)).$$

As $Pred(T_M, Q_M, L_M)$ outputs data of the same format as $Obs(T_S, Q_S, L_S)$, we can easily apply the same system properties on both data sets which enable us to investigate equivalence with respect to the set of system properties P. Formally, we define equivalence between a model and an implementation as:

Definition 4 *A model* M *is equivalent with respect to a system implementation* S *and a set of system properties P, iff:*

$$\forall p \in P : p(Obs(T_S, Q_S, L_S)) = p(Pred(T_M, Q_M, L_M))$$

This equivilance is denoted $S \equiv M$*.* □

Since the model is an abstraction of the system, it might be desired to have a certain amount of tolerance in the equivalence relation. This tolerance can however be encapsulated within the formulation of the system properties.

4.3 Using model equivalence for validation

The method for establishing an equivalence relation between a model and a system described in section 4.2 compares two data sets, one from the observation of the real system and one from the analysis of the model. If the two data sets are equal when comparing them, with respect to a set of system properties, they are equivalent, according to Definition 4.

In order to use the equivalence relation for validation of a model, a single observation is however not sufficient. Multiple observations of the system should

be used to get confidence in the validity of the model. This since a single observation of the system will probably only cover a minor subset of potential behaviors of the system, as mentioned in Section 3. The system might have many different modes of operation, with different temporal behavior. These modes must be identified and observations should be made in as many of these different situations as possible and included in the model. Comparing the model with the system in different situations can point out differences that only occur in some situations, i.e. a dependency that has been missed when constructing the model.

There are other reasons as well for basing a validation on multiple observations. One reason is if a certain transient scenario is of special interest when validating, e.g. the temporal behavior during a state transition in the system. In many cases it is only possible to capture one occurrence of the situation per observation, since the time it takes to put the system in the appropriate state that allows the scenario is often quite long, especially if it requires input from the user. Multiple observations can be used to capture several occurrences of the scenario and thus improve the confidence in the model.

Another reason for using multiple observations is if the memory available for the monitoring of a system is limited. In many embedded systems, not much memory is available for monitoring of extra-functional properties such as timing; it is very likely that at most only a few seconds of execution can be measured. If a longer observation is desired, i.e. more data, several shorter observations can be made instead.

To conclude this section, when performing a validation of a model, it is important to use multiple observations in order to observe as much as possible of the system behavior, but it is also important that different types of system properties are used for the comparison (as mentioned in Section 4.1), in order to compare as much as possible of the observed behavior of the system with the predictions based on the model. A third issue is to test different system alterations to verify that their impact on the model is the same as on the real system, i.e. to determine if the model is robust. In the next section, we will discuss how to use multiple model validations in order to analyze the model robustness.

5 Model Robustness

A model is *robust* with respect to a change in the implementation of the system if the change when applied to the model affects the predictions based on the model in the same way as it affects the observed behavior of the system. If a model is robust, it implies that the relevant behaviors and semantic relations are indeed captured by the model at an appropriate level of abstraction. In this section we propose a method for determining the robustness of a model in our framework. We refer to this activity as *sensitivity analysis*.

To exemplify the importance of model robustness, imagine a system containing a binary semaphore protecting a shared resource. A timeout occurs if a task has been waiting on the semaphore for a certain predefined time. If the

timeout occur, the execution time of the task is increased due to the error handling necessary. However, in all previous versions of the system, this timeout has never occurred. If the timeout is left out when constructing the timing model of the system the model still seems accurate since the timeout never occurs. However, as a result from changing the system, e.g. increasing the execution time of another task, the timeout will in some cases occur. Since the timeout was not captured in the model the system's behavior will diverge from the behavior predicted based on the model.

Our approach to sensitivity analysis is influenced by *system identification*. System identification is a technique used in the domain of control theory [2]. By measuring and observing the input-output relationship between signals in the process a model can be determined in terms of a transfer function. Validating models based on the system identification approach is somewhat related to testing. Typically, output signals predicted using the model is compared with the output signals of the physical process. Hence, the model is regarded as correct if the analysis and the physical process generate approximately the same output, if fed with the same input.

Testing the model with different input signals and comparing the prediction with the signals produced by the actual system is fine if the process is continuous in its nature. It is fair to assume that we can interpolate the behavior in between the tested signals. However, computer software is not continuous; they are discontinuous systems meaning that the behavior may change dramatically as a result of small changes in the system. A model of a software system can thus quickly become invalid when the system evolves, if the model is not robust with respect to typical changes. By analyzing the impact on the system caused by different changes, it is possible to determine if the model is sensitive to such changes, i.e. less robust.

5.1 Sensitivity Analysis

In this section, we will present how to analyze the robustness of a model using a sensitivity analysis. The basic idea is to test different alterations and verify that they affect the behavior predicted by the model in the same way as they affect the observed behavior of the system. First a set of *change scenarios* has to be elicitated. The change scenarios should be representative for the probable changes that the system may undergo. Typical examples of change scenarios are to change the execution times of a task or to introduce new types of messages on already existing communication channels. The change scenario elicitation requires, just as developing scenarios for architectural analysis, experienced engineers that can perform educated guesses about relevant and probable changes.

The next step is to construct a set of systems variants $\{S_1, ..., S_i\}$ and a set of corresponding models $\{M_1, ..., M_i\}$. The system variants $\{S_1, ..., S_i\}$ are versions of the original system, S_0, where i different changes have been made corresponding to the i different change scenarios. Note that these changes only needs to reflect the impact on the temporal behavior caused by the change scenarios, they do not have to result in any functional improvements of the

system. These changes are therefore easy to implement. The model variants are constructed in a similar way. $\{M_1, ..., M_i\}$ are the result of updating the initial model M_0 according to the same change scenarios.

Each model variant is then compared with its corresponding system variant by investigating if they are equivalent as defined in Definition 4. If all variants are equivalent, including the original model and system, we say that the model is robust. Formally we define robustness as follows:

Definition 5 *A model M is robust with respect to a system implementation S iff:*

$$\forall i S_i \equiv M_i$$

where $0 < i < N$ corresponds to a change scenario and N is the number of change scenarios. □

However, note that each comparison made to decide equivalence between a model and system variant should be made according to the recommendations presented in Section 4.3.

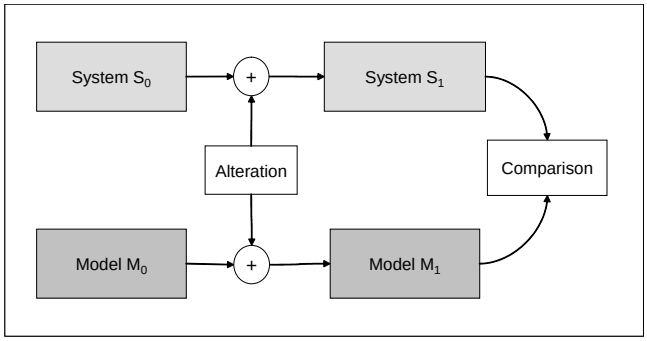

Fig. 2. Analyzing model robustness

In Figure 2 we have depicted the general process of analyzing the robustness of a model. An alteration, one of the identified change scenarios, is performed on the system S_0 and the model M_0 is updated to reflect the impact of the change. This results in a system variant S_1 and a model variant M_1, which are then compared as described in Section 4.3. If M_1 are equivalent to S_1 as defined in Definition 4, the model M_0 is robust with respect to that alteration.

6 Conclusion

In this paper we have addressed the problem of how to validate a model describing the temporal behavior of a large real-time system. We have proposed a

methodology for determining the equivalence between a timing model and the temporal behavior of the corresponding system, with respect to a set of system properties. Moreover, we have described the different types of such properties and we have also described how a sensitivity analysis can be used to study the robustness of a model. Further, different sources of errors in the model development process have been identified. We plan to test this approach in practice by applying it both on the case study described in the introduction, (the ABB robot controller) and also use it in another case study on a different system. Furthermore, we plan to investigate how the model constrution process can be facilitated. We believe that the model constrution process is the weakest link in this approach so automation of this part would be a major benefit.

References

1. O. Balci. Guidlines for Successful Simulation Studies. In *Proceedings of the 1990 Winter Simulation Conference*. Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 2061-0106, U.S.A., 1990.
2. R. Johansson. *System Modeling Identification*. ISBN 0-13-482308-7. Prentice-Hall, 1993.
3. A. M. Law and M. G. McComas. How to Build Valid and Credible Simulation Models. In *Proceedings of the 2001 Winter Simulation Conference*. Averill M. Law and Associates,Inc., P.O. Box 40996, Tucson, AZ 85717, U.S.A., 2001.
4. R. G. Sargent. Validation and Verification of Simulation Models. In *Proceedings of the 1999 Winter Simulation Conference*. Department of Electrical Engineering and Computer Science, College of Engineering and Computer Science, Syracuse University, Syracuse, NY 13244, U.S.A., 1999.
5. W. Schutz. On the Testability of Distributed Real-Time Systems. In *Proceedings of the 10th Symposium on Reliable Distributed Systems, Pisa, Italy*. Institut f. Techn. Informatik, Technical University of Vienna, A-1040, Austria, 1991.
6. M. E. Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *8th International Conference on Real-Time Computing Systems and Applications*. IEEE, March 2002.
7. A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-time Systems. In *Proceedings of RTCSA 03*. Department of Computer Science and Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, Sweden, 2003.
8. A. Wall, J. Andersson, and C. Norström. Probabilistic Simulation-based Analysis of Complex Real-time Systems. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing*. Department of Computer Science and Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, Sweden, 2003.