Embedded Software Analysis with MOTOR

Joost-Pieter Katoen[1], Henrik Bohnenkamp[1],
Ric Klaren[1], and Holger Hermanns[1,2]

[1] Faculty of Electrical Engineering, Mathematics and Computer Science,
University of Twente, The Netherlands
[2] Saarland University, D-66123 Saarbrücken, Germany

Abstract. This paper surveys the language MODEST, a Modelling and Description language for Stochastic and Timed systems, and its accompanying tool-environment MOTOR. The language and tool are aimed to support the modular description and analysis of reactive systems while covering both functional and non-functional system aspects such as hard and soft real-time, and quality-of-service aspects. As an illustrative example, the modeling and analysis of a device-absence detecting protocol in plug-and-play networks is described and is shown to exhibit some undesired behaviour.

1 Introduction

Background and motivation. The prevailing paradigm in computer science to abstract from physical aspects is gradually being recognized to be too limited and too restricted. Instead, classical abstractions of software that leave out "non-functional" aspects such as cost, efficiency, and robustness need to be adapted to current needs. In particular this applies to the rapidly emerging field of *"embedded"* software [14, 32].

Embedded software controls the core functionality of many systems. It is omnipresent: it controls telephone switches and satellites, drives the climate control in our offices, runs pacemakers, is at the heart of our power plants, and makes our cars and TVs work. Whereas traditional software has a rather transformational nature mapping input data onto output data, embedded software is different in many respects. Most importantly, embedded software is subject to complex and permanent interactions with their – mostly physical – environment via sensors and actuators. Typically software in embedded systems does not terminate and interaction usually takes place with multiple concurrent processes at the same time. Reactions to the stimuli provided by the environment should be prompt (timeliness or responsiveness), i.e., the software has to "keep up" with the speed of the processes with which it interacts. As it executes on devices where several other activities go on, non-functional properties such as efficient usage of resources (e.g., power consumption) and robustness are important. High requirements are put on performance and dependability, since the embedded nature complicates tuning and maintenance.

Embedded software is an important motivation for the development of modeling techniques that on the one hand provide an easy migration path for design engineers and, on the other hand, support the description of quantitative system aspects. This has resulted in various extensions of light-weight formal notations such as SDL (System Description Language) and the UML (Unified Modeling Language), and in the development of a whole range of more rigorous formalisms based on e.g., stochastic process algebras, or appropriate extensions of automata such as timed automata [1], and probabilistic automata [41]. Light-weight notations are typically closer to engineering techniques, but lack a formal semantics; rigorous formalisms do have such formal semantics, but their learning curve is typically too steep from a practitioner's perspective and they mostly have a restricted expressiveness.

The modeling formalism MODEST. This paper surveys MODEST, a description language that has a rigid formal basis (i.e., semantics) and incorporates several ingredients from light-weight notations such as exception handling[1], modularization, atomic assignments, iteration, and simple data types, and illustrates the accompanying tool support MOTOR by modeling and analyzing a device-absence detecting protocol in plug-and-play embedded networks.

MODEST is based on classical process algebra like CSP and CCS, and counts therefore as a *compositional* specification formalism: the description of complex behaviour is obtained by combining the descriptions of more simple components. Inherent to process algebra is the elegant way of specifying concurrent computations. MODEST is enhanced with convenient language ingredients like simple data-structures and a notion of exception handling. It is capable to express a rich class of non-homogeneous stochastic processes and is therefore most suitable to capture non-functional system aspects. MODEST may be viewed as an overarching notation for a wide spectrum of prominent models in computer science, ranging from labeled transition systems, to timed automata [1, 13] (and probabilistic variants thereof [31] and stochastic processes such as Markov chains and (continuous-time and generalised) Markov decision processes [22, 24, 35, 41].

Approach. With MODEST, we take a *single-formalism, multi-solution* approach. Our view is to have a single system specification that addresses various aspects of the system under consideration. Analysis thus refers to the same system specification rather than to different (and potentially inconsistent) specifications of system perspectives like in the UML. Analysis takes place by extracting simpler models from MODEST specifications that are tailored to the specific property of interest. For instance, for checking reachability properties, a possible strategy is to "distill" an automaton from the MODEST specification and feed it into an existing model checker such as SPIN [28] of CADP [19]. On the other hand, for carrying out an evaluation of the stochastic process underlying a MODEST specification, one may resort to discrete-event simulation, as, for instance, offered by the MÖBIUS tool environment.

[1] Exception handling in specification languages has received scant attention. Notable exceptions are Enhanced-LOTOS [21] and Esterel [6].

The tool-environment MOTOR. In order to facilitate the analysis of Modest specifications, the tool MOTOR [11] has been developed. Due to the enormous expressiveness of Modest, ranging from labeled transition systems to Markov decision processes and timed automata, there is no generic analysis method at our disposal that is able to cover all possible models. Instead, Motor aims at supporting a variety of analysis methods tailored to a variety of tractable sub-models. Our philosophy is to connect Motor to existing tools rather than implementing successful analysis techniques anew. Currently, connections to the CADP toolbox [19] and the multi/formalism - multi/solution Möbius tool environment [18] have been established. The former is aimed at assessing qualitative properties, whereas the latter is a performance evaluation tool supporting numerical methods and discrete-event simulation techniques. The case study described in this paper exploits the simulation facilities of Möbius.

Organization of this survey. Section 2 introduces the main syntactical constructs of Modest by means of modeling some example mutual exclusion protocols and presents its semantics by means of some examples. Section 3 briefly describes the Motor tool environment. Section 4 presents the modeling and analysis of a protocol in highly dynamic networked embedded systems and shows how this analysis reveals an undesired phenomenon. Section 5 finally concludes and gives some directions for future work. This paper is intended as a tutorial and does neither provide details of the syntax and semantics of Modest, nor the implementation details of Motor and the full details of the case study. Pointers to relevant papers where such details can be found are given in the various sections.

2 The Modeling Language MoDeST

2.1 Syntax

This section introduces the main syntactical constructs of Modest by means of modeling some example mutual exclusion protocols. The first one is a typical mutual exclusion algorithm where global variables are used to regulate the access to the critical section. The second algorithm uses timing to synchronize this access, whereas the latter is a randomized algorithm and does only guarantee mutual exclusion with a certain probability (that differs from one). A more detailed description of the Modest language can be found in [16].

Pnueli's mutual exclusion algorithm. The first example is a mutual exclusion protocol for two processes, called `P` and `Q`, due to Pnueli [38]. There is a single shared variable `s` which is either 0 or 1, and initially 1. Besides, each process has a local Boolean variable `y` that initially equals 0 and that may be inspected by the other process. The Modest specification of this algorithm is given below. Actions and assignments are the most elementary syntactical constructs in Modest. The global declaration part (cf. the first two lines) contains action, variable and constant (if any) declarations. In this paper, we adopt the convention that action names consist of two parts connected by an underscore. The model consists of the parallel composition of two processes as specified by the par-construct in the last four lines. Although in this example there is no communication between the processes via actions, the principle of par is that processes execute actions in common synchronously, and other actions autonomously. Such communication mechanism is rather common in process algebras such as CSP [27] and is a convenient mechanism for compositional modeling. A process description consists of an optional declaration part (absent in this example) of local actions and variables and a behaviour description, in this example consisting of a simple do-iteration for both P and Q. The statement s = false, y0 = true is a multiple assignment in which variable y0 is set to true and s to false in a single, atomic step. The when-statement may be read as "wait until". The other statements have the obvious meaning.

```
action enter_cs1, enter_cs2;
bool s = true, y0 = 0, y1 = 0;

process P() {
  do {
  :: {= s = false, y0 = true =};
     when (!y1 || s)
       enter_cs1;  // CS
     y0 = false    // leave CS
  }
}

process Q() {
  do {
  :: {= s = true, y1 = true =};
     when (!y0 || !s)
       enter_cs2;  // CS
     y1 = false
  }
}

par {
:: P()
:: Q()
}
```

The intuition behind this protocol is as follows. The variables y0 and y1 are used by each process to signal the other process of active interest in entering the critical section. On leaving the non-critical section, a process sets its own local variable y to 1. In a similar way this variable is reset to 0 once the critical section is left. The global variable s is used to resolve a tie situation between the processes. It serves as a logbook in which each process that sets its y variable to 1 signs at the same time. The test at the third line says that process P may enter its critical section if either y1 equals 0 – implying that its competitor is not interested in entering its critical section – or if s differs from 0 – implying that its competitor performed its assignment to y1 after P assigned 1 to y.

Fischer's timed mutual exclusion algorithm. As a second example we treat the mutual exclusion algorithm by Fischer [40] where time in combination with a shared variable is used to avoid processes to be in their critical section simultaneously. This algorithm is probably the most well-known benchmark example for real-time model checkers. Apart from the standard types bool, int and float for data, variables of the type clock can be used to measure the elapse of time. Clocks are set to 0 and advance implicitly, as opposed to ordinary data variables that need to be changed by means of explicit assignments. In the sequel we will use x and y to range over clock variables. All clocks run at the same pace. Clocks are a kind of alarm clocks that expire once they meet a value of type float. Such values can be of the usual form or can be samples from probability distributions (as we will see in the next example). Each process in Fischer's protocol has a single local clock `x` that is compared with the constant threshold values `d` and `k` to grab a ticket and check whether it is still the ticket's owner, respectively. The MODEST specification of the protocol is as follows.

```
int v;                                // ticket
action enter_cs, ...                  // action declarations

process P (int id) {                  // behaviour of a single thread
  clock x;                            // P's private timer
  const float k = 2.0, d = 1.0;
  do {
  :: when (v == 0) x = 0;             // once ticket is free, start timer
     do {
     :: when (x <= d)                 // wait for exactly k time units
          take_ticket {= v = id, x = 0 =};  // grab the ticket
        alt {
        :: when (v != id) x = 0;      // no longer own ticket
        :: when (v == id && x >= k) // ticket is still yours
             enter_cs;
           break {= v = 0 =}          // release ticket
        }
     }
  }
}

par {
:: relabel { take_ticket, enter_cs } by {take_ticket1,enter_cs1 } in P(1)
:: relabel { take_ticket, enter_cs } by {take_ticket2,enter_cs2 } in P(2)
}
```

A few remarks are in order. The `when`-statement that may guard an action (or assignment) may refer to data variables (like `v`) and clocks (e.g., `x`). In the latter case, the guarded action becomes enabled as soon as the condition in the `when`-clause becomes valid (and no other action becomes enabled at an earlier time instant). Note that the evaluation of the guards in a `when`-statement, the execution of the action and, if any, the mulitple assignments, is performed as a single atomic step, i.e., without interference of other parallel threads. This is similar to the well-known test-and-set principle [5][pp. 43] where the value of a shared variable is tested (i.e., a guard) and set (i.e., the assignment associated with an action) in a single step.

Remark 1. In this survey, we assume a maximal progress semantics that conforms to the semantics as taken by the discrete-event simulator of MOTOR-MÖBIUS. In case such maximal progress assumption is not adopted, an `urgent` clause may be used to force actions to happen at some time. This is similar to location invariants in timed automata [1, 13] and allows for the specification of non-deterministic timing. For instance, the following MODEST fragment:

```
clock x = 0;
urgent (x >= 75)
  when (x >= 20)
    enter_cs;
```

specifies that action `enter_cs` is enabled from 20 time units since resetting clock `x`, and that it should ultimately happen when `x` equals 75, as indicated by the `urgent`-clause. □

In Fischer's protocol, process `P`, for instance, waits until the global variable `v` – modeling a ticket that is needed to enter the critical section – equals zero and then sets the timer `x`. Subsequently, it waits exactly two time units before assigning P's `id` to `v` and is allowed to enter its critical section only when `v` still equals its `id`. In case it does not own the ticket, it has to wait again. The choice between `v == 0` and `v == id` is syntactically represented by the `alt`-construct that also allows for modeling non-deterministic choices. Recall that in case none of these conditions is met, the process waits. On leaving the critical section, the ticket is released and the entire procedure starts again. Note that the entire system is composed of two processes that are obtained from the blueprint `P` by instantiating it with the ids one and two and relabeling the actions in an appropriate way in order to avoid unintended synchronizations. By means of relabeling, actions are renamed in the behaviour expression, e.g., `rename a by b in P` will result in a process that behaves like `P` except that any syntactic occurrence of `a` in `P` is renamed into `a`.

Randomized Fischer's mutual exclusion algorithm. The last example is a randomized variant of Fischer's algorithm due to Gafni and Mitzenmacher [20]. The main difference is that the main activities in the protocol, such as inspecting the global variable `v`, and entering the critical section, are governed by exponential (or gamma) distributions. A MODEST specification of the case with gamma distributions is as follows.

```
int v = 0;     // ticket

process P (int id) {
clock x;
float k;
do {
:: when (v == 0) {= k = GAMMA(...), x = 0 =};
   do {
   :: when ( (x == k) && (v == 0) )
        take_ticket; {= v = id, k = GAMMA(...), x = 0 =};
      alt {
      :: when ( (v != id) ) break
      :: when ( (x == k) && (v == id) )
           enter_cs {= k = GAMMA(...) =};
         when (x == k) break {= v = 0 =}
      }
   }
}
}
```

```
par {
:: relabel { take_ticket, enter_cs } by {take_ticket1, enter_cs1} in P(1)
:: relabel { take_ticket, enter_cs } by {take_ticket2, enter_cs2} in P(2)
}
```

Other randomizedmutual exclusion protocols in [20] can be obtained in a similar way. Note that the value of variable `k` is determined by sampling a gamma-distribution. By requiring `x == k` in the `when`-clauses, it is enforced that the amount of time that has elapsed is indeed governed by a gamma-distribution.

Other syntactical constructs. It remains to explain the `palt`-construct that is used to model probabilistic choice. A `palt`-statement is in fact an action that has several alternative (multiple) assignments that can take place with accompanying successive statements. The likelihood of these alternatives is determined by weights. For instance,

```
take_ticket palt {
            :1: {= v = id, x = 0 =} P(v)
            :3: {= v = 0, x = 0 =} P(v)
            }
```

specifies that on the occurrence of action `take_ticket`, `v` will be set to `id` with probability $\frac{1}{1+3} = \frac{1}{4}$ and to zero with probability $\frac{3}{4}$. In both cases, `x` is reset. Note that the occurrence of the action, the resolution of the probabilistic choice, and the multiple assignments are executed atomically. In fact, an ordinary action occurrence with some multiple assignments can be viewed as syntactic sugar for a `palt`-statement with a single alternative.

As the case study furtheron does not use exception handling, we refrain from introducing this operator here.

Remark 2. The syntax of the control and data structures in MODEST is very similar to that of PROMELA, the protocol modeling language that is used as input language to the model checker SPIN [28]. For instance, similar constructs to `do`, `alt`, `when`, multiple assignments and `process` exist in PROMELA. There are, however, some differences. As PROMELA is aimed at describing protocols, communication channels and primitives to send and receive messages along them are first-class citizens in the language. In MODEST such communication buffers need to be modeled explicitly as separate processes. PROMELA incorporates an `atomic`-statement in which a sequence of statements can be executed atomically, i.e., without interference of other parallel processes; MODEST only supports multiple assignments. The main add-ons in MODEST are: the possibility of specifying discrete probabilistic branching (using `palt`) [2], real-time, and randomizedtime delays. Besides, the formal semantics of MODEST (see below) provides an unambiguous interpretation. □

[2] A similar construct has recently been suggested in a probabilistic variant of PROMELA [3].

2.2 Semantics

Stochastic timed automata. The MODEST semantics is defined in terms of an extension of timed automata. The extension is needed to accommodate for the `palt`-construct and the random delays. Whereas timed automata are aimed to finitely represent infinite-state real-time systems, our variant – baptized stochastic timed automata – focuses on finitely representing stochastic timed systems. (As for timed automata, the underlying interpretation of such models is indeed an infinite-state, infinitely branching structure.) In case a MODEST specification does not cover any probabilistic choice, the semantics obtains the symbolic automata one intuitively expects (cf. Fig. 1). In these automata, transitions are

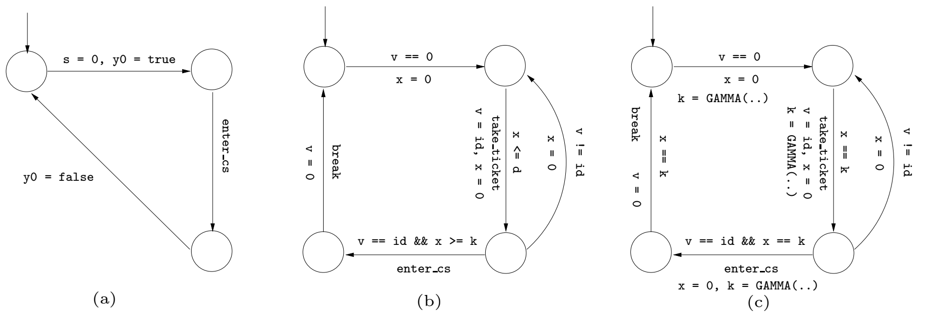

Fig. 1. Automata for a single thread in (a) Pnueli's, (b) Fischer's and (c) randomized Fischer's mutex algorithm.

equipped with an action (that may be subject to interaction with other parallel processes), a guard and a multiple assignment. All these attributes are optional with the condition that either an action or an assignment is present. An action is absent for statements like `x = 0;`, a guard is absent (i.e., true) in absence of a `when`-statement, and a (multiple) assignment is absent for actions without accompanying assignments. If actions synchronize, the resulting transition is decorated with the common action, the joined multiple assignments (provided they do not assign to the same variables), and the conjunction of guards.

To treat discrete probabilistic branching, the concept of transition is refined cf. Fig. 2. A transition is a one-to-many edge labeled with an action and a guard (as before), but where different multiple assignments are possible, and where each alternative has an associated weight to determine the likelihood. The intuitive interpretation of the simple stochastic timed automaton in Fig. 2, for instance, is as follows. Once the conditions `v != id` and `x == k` hold (and the environment is able to participate, if needed, in action `take_ticket`), both outgoing transitions of state s are enabled, and one of them is non-deterministically chosen. On selecting the rightmost transition, action `take_ticket` is performed, and there are two possible successor states. With probability $\frac{1}{4}$ the assignments `v = id` and `x = 0` are performed (atomically) and the automaton moves to state u, while with the remaining probability $\frac{3}{4}$, `x` and `v` are both reset to zero, and the next state equals t. When the leftmost transition is chosen, there is a single alternative (i.e., a probabilistic choice with one alternative that occurs with probability one).

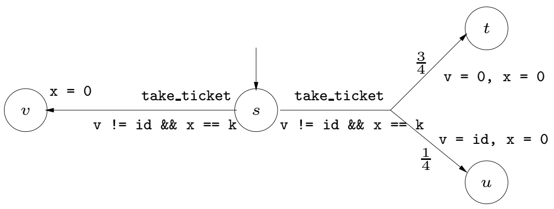

Fig. 2. A simple stochastic timed automaton.

Operational semantics. The mapping from Modest onto stochastic timed automata is defined by means of structured operational semantics [34]. This semantics is a simple adaptation of the usual operational semantics of programming languages and process calculi. The slight adaptation is needed to accommodate probabilistic choices. Let us consider some examples. The standard inference rule for actions

$$a \xrightarrow{a,tt} \surd$$

states that action a can perform execute a at any time (i.e., guard is true), and evolves into the successfully terminated behaviour $\surd$. Note that $\surd$ is not a syntactical entity, but is just used to define the semantics. For Modest, this rule is written as:

$$a \xrightarrow{a,tt} \mathcal{P}$$

where $\mathcal{P}$ is a trivial distribution such that $\mathcal{P}(\varnothing, \surd) = 1$, i.e., the probability of performing no assignments and evolving into $\surd$ equals one. The inference rule for Modest actions is thus a simple generalization of the standard inference rules for actions. The target of a transition for Modest is no longer an expression (like $\surd$) but a *probability space* with accompanying probability measure (e.g., a trivial distribution). The same applies to the other operators. For instance, for alternative composition, the standard inference rule

$$\frac{P_i \xrightarrow{a,g} P_i' \qquad (1 \leqslant i \leqslant k)}{\mathsf{alt}\{:: P_1 \ \ldots \ :: P_k\} \xrightarrow{a,g} P_i'}$$

stating that whenever the alternative P_i can make an a, g-move, then the alternative composition can do so as well, is generalized yielding:

$$\frac{P_i \xrightarrow{a,g} \mathcal{P}_i \qquad (1 \leqslant i \leqslant k)}{\mathsf{alt}\{:: P_1 \ \ldots \ :: P_k\} \xrightarrow{a,g} \mathcal{P}_i}$$

Note, again, that the target of an expression is a probability space, viz. $\mathcal{P}_i$.

The semantics of the new operators is defined as follows. For probabilistic choice the inference rule is

$$a\ \mathsf{palt}\ \{:w_i:\ A_i\ ;\ P_i\}_{i \in I} \xrightarrow{a,tt} \mathcal{P} \text{ with } \mathbf{P}(A_i, P_i) = \frac{w_i}{\sum_{j \in I} w_j}$$

where, for simplicity, it is assumed here that all P_i's are distinct and where $\mathbf{P}$ is the probability measure of probability space $\mathcal{P}$. It is, in fact, an alternative composition in which one of the alternatives (i.e., P_i) and the associated multiple assignments (i.e., A_i) is chosen with a certain probability that is determined by the weights. Here, it is assumed that all weights (i.e., w_i) are strictly positive. Weights can either be constant, like in our examples, but may also be an expression[3]. Guards as specified by when-statements can be handled easily:

$$\frac{P \xrightarrow{a,g} \mathcal{P}}{\mathsf{when}(b)\ P \xrightarrow{a,b \wedge g} \mathcal{P}}$$

where b is a boolean expression. Thus, if P can perform action a with guard g, $\mathsf{when}(b)\ P$ can perform a with guard $b \wedge g$. The semantics of do-statements, relabeling, breaks, and process instantiation are standard and omitted here. We conclude with parallel composition. First, note that:

$$\mathsf{par}\{::\ P_1\ \ldots\ ::\ P_k\} \stackrel{\text{def}}{=} (\ldots((P_1 \parallel_{B_1} P_2)\ldots)) \parallel_{B_{k-1}} P_k$$

where $\parallel_B$ is CSP-like parallel composition [27] and

$$B_j = \left(\bigcup_{i=1}^{j} \alpha(P_i) \right) \cap \alpha(P_{j+1})$$

where $\alpha(P)$ denotes the set of actions that P can be involved in. This observation allows us to define the semantics of the `par`-construct in terms of $\parallel_B$ where B is the common alphabet of all processes put in parallel. The inference rule for standard CSP parallel composition for executing autonomous actions, i.e., actions that are not subject to any interaction with other parallel processes, is defined as follows:

$$\frac{P \xrightarrow{a,g} P' \text{ and } P' \neq \surd}{P \parallel_B Q \xrightarrow{a,g} P' \parallel_B Q} \text{ and } \frac{P \xrightarrow{a,g} \surd}{P \parallel_B Q \xrightarrow{a,g} Q \backslash B}$$

where $Q \backslash B$ equals behaviour Q except that actions in the set B are prohibited. This conforms to the idea that Q should synchronize on such actions with P, where P is impossible to do so. For MODEST, these rules are generalized towards:

$$\frac{P \xrightarrow{a,g} \mathcal{P} \qquad a \notin B}{P \parallel_B Q \xrightarrow{a,g} \mathcal{R}} \quad \text{with} \quad \begin{array}{l} \mathbf{R}(A, P' \parallel_B Q) = \mathbf{P}(A, P') \\ \mathbf{R}(A, Q \backslash B) \quad = \mathbf{P}(A, \surd) \end{array}$$

[3] The semantics of the latter case is more involved and omitted here.

For synchronizations, the inference rule is readily obtained from the inference rules for CSP-synchronization:

$$\frac{P \xrightarrow{a,g} \mathcal{P} \qquad Q \xrightarrow{a,g'} \mathcal{Q} \qquad a \in B}{P \parallel_B Q \xrightarrow{a,g \wedge g'} \mathcal{R}}$$

where the probability space $\mathcal{R}$ is the product of $\mathcal{P}$ and $\mathcal{Q}$ defined by

$$\mathbf{R}(A \cup A', P' \parallel_B Q') \;=\; \mathbf{P}(A, P') \cdot \mathbf{Q}(A', Q')$$

in case both P and Q do not successfully terminate and A and A' do not assign values/expressions to the same variables. If one of these processes successfully terminates, a slight modification of this equation applies, cf. [16]. In case P and Q perform (possibly inconsistent) assignments, an exception is raised.

Interpretation of stochastic automata. The interpretation of timed automata is typically defined in terms of infinite-state timed transition systems. For stochastic timed automata this is done in a similar way. A configuration in such transition system records the current state of the stochastic timed automaton and the valuation of all (data and clock) variables. If $s \xrightarrow{a,g} \mathcal{P}$ and the current valuation satisfies guard g, then with probability $\mathbf{P}(A, s')$, where $\mathbf{P}$ is the probability measure of $\mathcal{P}$, the valuation is changed according to the sequence of assignments A, and the next state is s'. Under the maximal progress assumption, time is advanced with some positive amount $d > 0$ if in the current state no other outgoing transition is enabled at some time instant $d' < d$. The advance of time with d means that all values of clock variables are increased by d while letting all other variables unchanged. Note that this interpretation yields a continuous space model with infinitely many states and infinite branching. For a more detailed description of this semantics we refer to [9].

3 The Tool Environment MOTOR

The case study assessed in this paper has been analyzed by means of the Modest tool environment Motor and the performance evaluation environment Möbius. In this section, we will briefly discuss these two tools.

Möbius. This is a performance evaluation tool environment developed at the University of Illinois at Urbana-Champaign, USA [18]. Möbius supports multiple input formalisms and several evaluation approaches for these models. Fig. 3 (a) shows an overview over the Möbius architecture. Atomic models are specified in one of the available input formalisms. Atomic models can be composed by means of state-variable sharing, yielding so called composed models. Notably, atomic models specified in different formalisms can be composed in this way. This allows to specify different aspects of a system under evaluation in the most suitable formalism. Along with an atomic or composed model, the user specifies a reward model, which defines a reward structure on the overall model.

On top of a reward model, the tool provides support to define experiment series, called *Studies*, in which the user defines the set of input parameters for which the composed model should be evaluated. Each combination of input parameters defines a so-called *experiment*. Before analyzing the model experiments, a solution method has to be selected: MÖBIUS offers a powerful (distributed) discrete-event simulator, and, for Markovian models, explicit state-space generators and numerical solution algorithms. It is possible to analyze transient and steady-state reward models. The solver solves each experiment as specified in the *Study*. Results can be administered by means of a database.

The different components constituting a solvable model are specified by means of a series of editors written in Java. Transparent to the user, models are translated into C++ code, compiled and linked together with the necessary supporting libraries, building an executable. The control over build and run of the solver is again done from a Java component. MÖBIUS currently supports four input formalisms: Bucket and Balls (an input formalism for Markov Chains), SAN (Stochastic Activity Networks) [33, 39], and PEPA (a Markovian Stochastic process algebra) [26]. Recently, the MODEST modeling language has been integrated into the MÖBIUS framework.

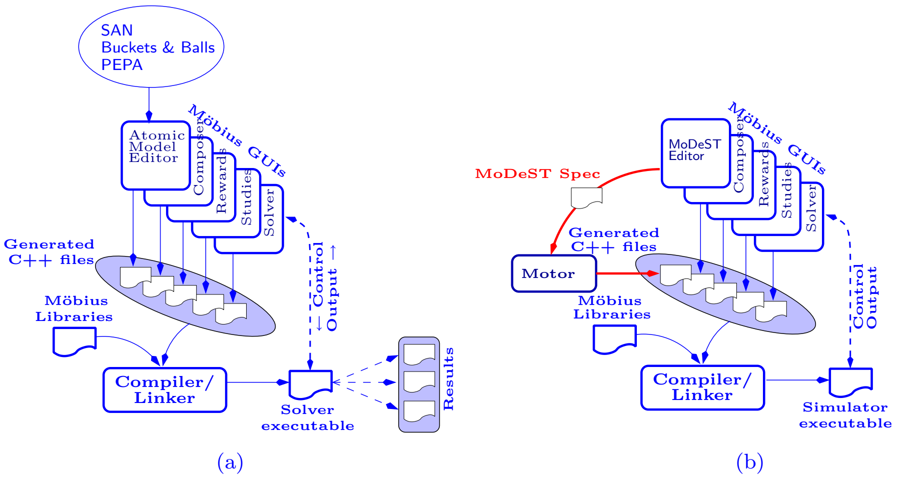

Fig. 3. MÖBIUS Architecture and MOTOR integration.

MOTOR. In order to facilitate the analysis of MODEST models, we have developed the prototype tool MOTOR [11]. MODEST is a very expressive language, covering a wide range of timed, probabilistic, nondeterministic, and stochastic models. The spectrum of covered models includes ordinary labeled transition systems, discrete and continuous time Markov chains and decision processes, generalized semi-Markov processes, and timed and probabilistic timed automata. These submodels play a crucial role in the context of MOTOR. The enormous expressiveness of Modest implies that no generic analysis algorithm is at hand. Instead, Motor aims at supporting a variety of analysis algorithms tailored to the variety of analyzable submodels. The philosophy behind Motor is to connect Modest to existing tools, rather than re-implementing existing analysis algorithms anew. The advantages of this approach are *(i)* that excellent work of leading research groups is made available for the analysis of Modest models, and *(ii)* that this is achieved with only moderate implementation effort. This requires a well-designed interfacing structure of Motor, which is described in [11].

The first tool Modest was connected to was the CADP toolbox [19]. The latter is a widespread tool set for the functional design and verification of complex systems.To complement the qualitative analysis of Modest specifications using CADP we started joint efforts with the Möbius developers [8] to link to the powerful solution techniques of Möbius for quantitative assessment. The main objective was to simulate Modest models by means of the Möbius distributed discrete-event simulator, because a stochastic simulator can cope with one of the largest class of models expressible in Modest.

Motor *and* Möbius. The integration of Modest into Möbius is done by means of Motor. For this integration, Motor has been augmented with a Modest-to-C++ compiler. From a user-perspective, the Möbius atomic model interface to design Modest specifications is an ordinary text editor. Whenever a new version of the Modest specification is saved to disk the Motor tool is called automatically in order to regenerate all C++ files (cf. Fig. 3 (b)). Additionally, a supporting C++ library has been written for Möbius, which contains two components: first, a virtual machine responsible for the execution of the Modest model, and second, an interface to the simulator of Modest.

As with all other types of atomic models of Möbius is it possible to define reward models and studies on top of Modest models. The state variables which are accessible for reward specification are the *global variables* of the Modest specification. Additionally, it is possible to declare constants in the Modest specification as *extern*, meaning that these constants are actually input parameters of the model, pre-set according to the specified study.

Due to the possibility to specify non-Markov and non-homogeneous stochastic processes, only simulation is currently supported as a suitable evaluation approach for Modest models within Möbius. While it is in principle possible to identify sublanguages of Modest corresponding to Markov chain models, this has not been implemented in Motor yet.

4 Case Study: Distributed Device-Absence Checking

As an illustrative example application of our framework and tool, we consider the modeling and subsequent analysis of a protocol that is aimed to maintain (and to some extent disseminate) up-to-date information about the presence (or absence) of nodes in a dynamically changing distributed environment. That is to say, the protocol allows for the monitoring of the availability of a node by other nodes. Normally, when a node goes off-line, it informs other nodes by sending a bye-message, but if it suddenly becomes unavailable, no such indication is sent, and the studied protocol comes into play. Important requirements on the protocol are that it should be able to detect the absence of nodes fast (i.e., within about a second) while avoiding to overload devices. Related protocols to this absence-checking protocol [7], nicknamed "ping protocol" in the sequel, are failure detection and monitoring protocols. Failure detection protocols [36, 37] aim to identify whether in a group of nodes, one or more nodes stop executing correctly. In the ping protocol there are two types of nodes, however, only the failure of a single type of node is relevant. Monitoring protocols involve the aggregation of various sorts of data (such as availability information) that are distributed among nodes in the network [30]. The considered protocol is intended as an enhancement to node (or service) discovery protocols that are common in "plug-and-play" distributed systems to find nodes. It is self-organizing in the sense that it continues to operate properly without manual intervention under the – according to different patterns – joining and (un)intentional leaves of nodes.

4.1 The Ping Protocol

Here, we summarize the behaviour of the ping protocol [7]. The protocol originally has been developed as an extension of the service discovery protocol in the UPnP standard (Universal Plug and Play), but may also be used as extension of similar protocols such as SLP, Rendezvous and Jini.

Two types of nodes are distinguished: simple nodes (*devices*) and somewhat more intelligent ones, called *control points* (CPs). The basic protocol mechanism is that a CP continuously probes (i.e., pings) a device that replies to the CP whenever it is (still) present. The essence of the protocol is to automatically adapt the probing frequency in case a device tends to get over- or underloaded. This self-adaptive mechanism is governed by a couple of parameters that are described in more detail furtheron. The CPs are dynamically organized in an overlay network by letting the device, on each probe, return the ids of the last two processes that probed it as well. On the detection of the absence of a device, this overlay network is used to rapidly inform CPs about the leave of the device. For the sake of simplicity, the latter information dissemination phase of the protocol is not considered here.

Device behaviour. A device maintains a probe-counter *pc* that keeps track of the number of times the device has been probed so far. On the receipt of a probe, this counter is increment by the natural Δ, which typically equals one, and a reply is sent to the probing CP with as parameters the (just updated) value of *pc*, and the ids of the last two distinct CPs that probed the device. The latter information is needed to maintain the overlay network of CPs[4], whereas

[4] By returning two distinct CP ids, the overlay network forms a tree with depth $\log_2 N$ where N is the number of CPs, with a high probability.

the returned value of *pc* is used by CPs to estimate the load of the device. As Δ is device-dependent, and typically only known by the device itself, a CP cannot distill the actual probing frequency of a device, but only its own perceived probing frequency. Note that Δ can be used by a device to control its load, e.g., for a larger Δ, CPs consider the device to be more (or even over-) loaded sooner, and will adjust (i.e., lower) their probing frequency accordingly resulting in a lower deviceload. Although in principle the value of Δ can be changed during the lifetime of a device, in the sequel we assume it to be constant.

CP behaviour. The behaviour of a CP is more intricate. The basic mechanism for communicating with a device is a bounded retransmission protocol (à la [17]): a CP sends a probe ("are you still there?"), and waits for a reply. In absence of a reply, it retransmits the probe. Otherwise, the CP considers the reply as a notification of the (still) presence of the device, and continues its normal operation. Probes are retransmitted maximally three times. If on none of the four probes a reply is received, the CP considers the device to have left the network, and starts to disseminate this information to other CPs using the overlay network. The protocol allows to distinguish between the timeout value TOF after the first probe and the timeout value after the other (maximally three) probes TOS. Typically, $\mathsf{TOS} < \mathsf{TOF}$.

Let us now consider the mechanism for a CP to determine the probing frequency of a device. Let δ be the delay between two consecutive, i.e., not retransmitted, probes. For given constants δ_{min} and δ_{max} with $\delta_{max} \gg \delta_{min}$, the CP has to obey

$$\delta_{min} \leqslant \delta \leqslant \delta_{max}.$$

The value of δ is adapted after each successful probe in the following way. Assume the CP sends a probe to the given device at (its local) time t and receives a reply on that with probe-count *pc*. (In case of a failed probe, the time at which the retransmitted probe has been sent is taken.) The next probe is sent at time $t' > t$, and let *pc'* be its returned probe-count. $t'-t$, thus, is the time delay between two successive successful probes. The probeload of the device, as perceived by the CP, is now given as

$$\gamma = \frac{pc' - pc}{t' - t}.$$

The actual probeload of the device equals γ/Δ. For given maximal and minimal probeloads γ_{max} and γ_{min} for the CP, and constant factors $\alpha_{inc}, \alpha_{dec} > 1$, the delay δ is adapted according to the following scheme, where δ' and δ refer to the new and previous value of δ, respectively:

$$\delta' = \begin{cases} \min\big(\alpha_{inc} \cdot \delta, \delta_{max}\big) & \text{if } \gamma > \gamma_{max} \\ \max\big(\frac{1}{\alpha_{dec}} \cdot \delta, \delta_{min}\big) & \text{if } \gamma < \gamma_{min} \\ \delta & \text{otherwise} \end{cases}$$

This adaptive scheme is justified as follows[5]. In case the just perceived probeload γ exceeds the maximal load γ_{max}, the delay is extended (by a factor $\alpha_{inc} > 1$) with the aim to reduce the load. As δ should not exceed the maximal delay δ_{max}, we obtain the first clause of the above formula. This rule thus readjusts the probing frequency of a CP in case the number of CPs (probing the device) suddenly increases. If γ is too low, the delay is shortened in a similar way while obeying $\delta_{min} \leqslant \delta$. The second rule thus readjusts the probing frequency of a CP in case the number of CPs (probing the device) suddenly decreases. In all other cases, the load is between the maximal and minimal load, and there is no need to adjust the delay. Note that the maximal frequency at which a CP may probe a device – given that the protocol is in a stabilized situation – is given by $\max(\frac{1}{\delta_{min}}, \gamma_{max})$. The maximal actual probing frequency of a device is Δ^{-1} times this quantity.

4.2 Modeling in MoDeST

The ping protocol can be modeled in Modest in a rather straightforward manner. The entire specification consists of the parallel composition of a number of CPs, a number of devices and a network process. By making a precise description of the ping protocol in MoDeST, some small unclarities in the original protocol specification [7] were revealed, such as, e.g., the way in which the ids of the last two (distinct) probing CPs were managed.

As the main aim of our simulation study is an assessment of the self-adaptive mechanism to control the device's probe frequency, the devices are supposed to be present during the entire execution of the simulation (i.e., they are static), whereas the CPs join and leave the network frequently (i.e., they are highly dynamic). In order to govern the leave- and join-pattern of CPs, a separate process is put in parallel to the CPs that synchronizes on `join` and `leave` actions while timing these actions according to some profile as specified in the simulation scenario at hand (see below). For simplicity we omit these actions from the model of the CP as presented below.

The network is modeled as a simple one-place buffer. A shared variable `m` contains the current message in transit (if any) and has fields that contain the various parameters, e.g., `m.src` indicates the address of the source of `m`, `m.lck` indicates whether the structure contains a message, and `m.pc` is the probe counter of a reply message. As the only messages (in our model) from devices to CPs are replies, and from CPs to devices are probes, there is no need to distinguish message types.

To give an impression of the Modest specification of the ping protocol, we present the (basic, i.e., somewhat simplified) models of a device and CP. The ids of CPs in reply-messages and the bookkeeping of these ids by the device are omitted here, as the dissemination phase is not further considered. The basic behaviour of a device is modeled as follows:

[5] To avoid clustering of CPs, in fact, a CP adds a small value to δ' that is randomly determined. For the sake of simplicity, this is not described any further here.

```
process Device (int id) {
  action handle_probe, send_reply ;    // action declarations
  const int Delta = 1.0;               // probe increase
  clock x;                             // timer for reply time
  int pc = 0,                          // probe counter
      cp;
  float rpldel;                        // reply delay

  do {                                 // actual behaviour
  :: when ( (m.lck) && (m.dst == id) )
       handle_probe {=
         pc += Delta, cp = m.src, m.lck = 0,
         rpldel = min + (max - min)*EXP(...), x = 0 =};
     when ( x >= rpldel )              // rpldel time-units elapsed
       send_reply {= m.src = id, m.dst = cp, m.pc = pc =}
  }
}
```

Here it is assumed that the processing time of the device, i.e., the time between the receipt of a probe and transmitting its reply, is governed by an exponential distribution (see also below), but this could, of course, be any other reasonable distribution. Note that on simulating this model, the maximal progress assumption is adopted, i.e., on the expiration of the delay `rpldel` in the device, a reply is sent immediately. No further delay takes place. The basic behaviour of a CP is modeled as follows:

```
process CP (int id, ) {
  action send_probe, ....              // action declaration
  clock x;                             // timer for timeouts and delays
  int pc = 0,                          // probe counter of last reply
      i = 0;                           // probe counter
  float d = d_max,                     // delay until next probe (=delta)
        to,                            // timeout value
        pl;                            // pingload (= gamma)

  do {
  :: send_probe {= i++, m.src = id, m.dst = dev_id, x = 0, to = TOF =};
     do {                               // wait for reply or timeout
     :: alt {
        :: // timeout and more probe retransmissions allowed
           when ( (x >= to) && (i < 4) )
             send_probe
                  {= i++, m.src = id, m.dst = dev_id, x = 0, to = TOS =};
        :: // reply received in time
           when ( (x < to) && (m.lck) && (m.dst == id) )
             handle_reply {= m.lck = 0, pl = (m.pc - pc)/d, pc = m.pc =};
             alt {                        // adapt delay-to-ping
             :: when (pl > gamma_max)
                  alt { :: when (d * a_inc <= d_max)
                                {= d = d * a_inc =}
```

```
                      :: when (d * a_inc > d_max)
                           {= d = d_max =}
           :: when (pl < gamma_min)
                alt { :: when (d * 1/a_dec > d_min)
                           {= d = d * 1/a_dec =}
                      :: when (d * 1/a_dec <= d_min)
                           {= d = d_min =} }
           :: when ((pl >= gamma_min) && (pl <= gamma_max)) tau  // nop
           };
           x = 0;               // reset timer
           when ( x >= d )
             i = 0;
             break                         // restart probing
      :: // timeout and no retransmissions further allowed
         when ( (x >= to) && (i == 4) )
           dev_abs {= i = 0, pc = 0 =};   // signal device absence
           break                         // restart probing
      }
  }
}
```

On each outermost iteration, a CP starts by sending an initial probe to the device. As the first waiting time until a reply equals TOF, the variable `to` is set to that value, and clock `x` is reset. In the innermost iteration, there are three possibilities. In case the timer expires, signaling that the reply did not come in time, and the number of probe-transmissions did not exceed the maximum, the probe is retransmitted. Note that in this case `to` is set to TOS. If the timer expires, and the probe has been sent a maximal number of times, the device is assumed to have left the network and the dissemination phase is started. This is modeled in an abstract way using action dev_abs, and an immediate restart of the probing. In case a reply is received in time, the probeload is determined, the time until the next probe is determined, and probing is restarted after this delay `d`.

The last component of the MODEST specification is the model of the network. As stated before, the network is modeled as a one-place buffer for simplicity. Its model is as follows:

```
process Network () {
  action get_msg, ...       // action declarations
  clock x;                  // timer for message delay
  const int ploss = 1;      // message loss probability
  float del;                // random message delay

  do {
  :: when (m.lck != 0)
      get_msg {= m.lck = 0, x = 0, del = min + (max - min)*EXP(...) =};
      tau palt {
          :ploss: lose_msg
          :(10000 - ploss): alt {
                            :: when (x >= del) put_msg {= m.lck = 1 =}
```

```
                          :: when (m.lck != 0) lose_msg  // m overwritten
                            }
           }
  }
}
```

4.3 Analysis with MOTOR

To get insight into the behaviour of the probe protocol, in particular, into the self-adaptive mechanism to control the probe frequency of a device, the MoDeST model has been analyzed by means of discrete-event simulation using the MOTOR-Möbius interface. The main aim of the analysis was to obtain indications for reasonable values of the parameters of the protocol, in particular of α_{inc}, α_{dec} and TOF and TOS. The original protocol description [7] indicates that $\alpha_{inc} = 2$ and $\alpha_{dec} = \frac{3}{2}$ are appropriate choices, but leaves the other parameters unspecified.

A simulation scenario. To enable a simulation, a description of the configuration of the network (i.e., the number of CPs and devices) and their join- and leave-behaviour need to be given. We consider a configuration consisting of a single device and eight CPs. As our aim is to study the self-adaptive mechanism to control the probe frequency of a device, the device is assumed to be present during the entire simulation whereas the CPs have a more dynamic nature. Two CPs are continuously present, and six CPs join in a bursty fashion, one shortly after the other, are present for a short while, and then four suddenly leave altogether. The four then repeatedly all join and leave, until at some point in time all six CPs leave.

Stochastic assumptions. Various stochastic phenomena of the ping protocol have been modeled such as the transit delay of a message through the network and the connection time of a CP. To give an impression of the assumptions that have been made, we consider a few examples. The device response time, i.e., the delay that is exhibited by a device between receiving a probe and sending its reply is determined by

$$t_{min} + (t_{max} - t_{min}) \cdot p$$

where t_{min} and t_{max} equal 0.06 and 20 msec, respectively, and p is governed by a negative exponential distribution (with rate $\lambda = 3$). The one-way message delay for a fixed network (i.e., Ethernet-like) is determined in a similar way using $t_{max} = 1$, $t_{min} = 0.25$ and $\lambda = 3$, whereas a constant loss probability of 10^{-5} is assumed[6]. The connection times of CPs is chosen to be deterministic, but different for the various CPs.

[6] These probability distributions are not intended to reflect the actual delays or loss probabilities, but are merely used as indications. More precise indications are manufacturer specific (and not publicly available).

Simulation parameters. The following parameters exemplify the kind of information that is obtained from a simulation of the MODEST model of the ping protocol:

- N_{msg}, the average number of probes and replies exchanged per second
- P_{false}, the probability that a present device is considered to be absent
- P_{late}, the probability that the time until the next probe is exceeding some predefined maximum (e.g., 0.7 seconds), and
- T_{abs}, the average time until a CP detects the absence of a device.

In order to obtain these measures, the MODEST specification may be equipped with additional variables with the sole purpose of information gathering. For instance, in order to estimate P_{false}, the CP-model is extended with a counter that is incremented when a device is considered to be absent, cf. action `dev_abs` in the earlier presented model. In a similar way, the model of the device is enriched with a clock that measures the amount of time a probe is arriving too late, i.e., after the deadline of 0.7 seconds.

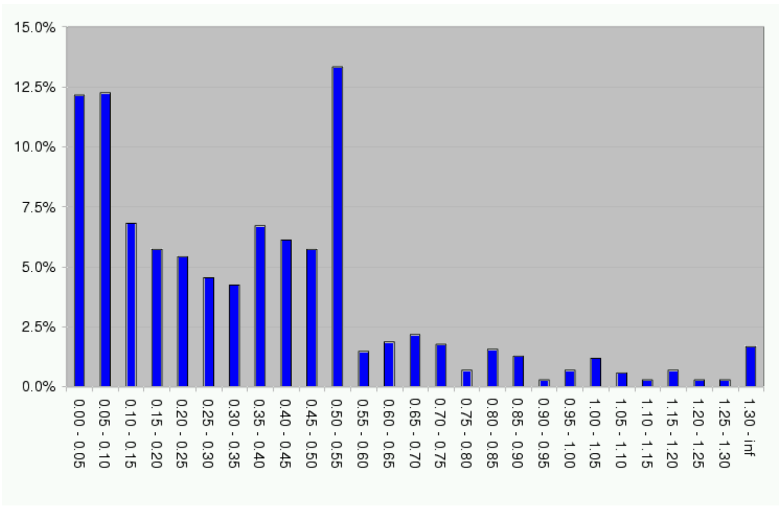

Fig. 4. Distribution of the delay between successive probes.

Some analysis results. To give an impression of the kind of results that can be obtained using the MOTOR tool we present three curves. The first plot (cf. Fig. 4) indicates the probability distribution of the probe delay as perceived at a device. P_{late} equals 0.048 and B equals 5.535 mps (messages per second); the average time-to-ping is about 0.32 seconds. The protocol was run using the values for α_{inc}, γ_{max}, TOF and so on, as indicated in [7] and as summarized in the first row of Table 1.

In our simulation study we concentrated on determining the effect of the values of α, γ and the timeout values TOF and TOS. Fig. 5, for instance, depicts the

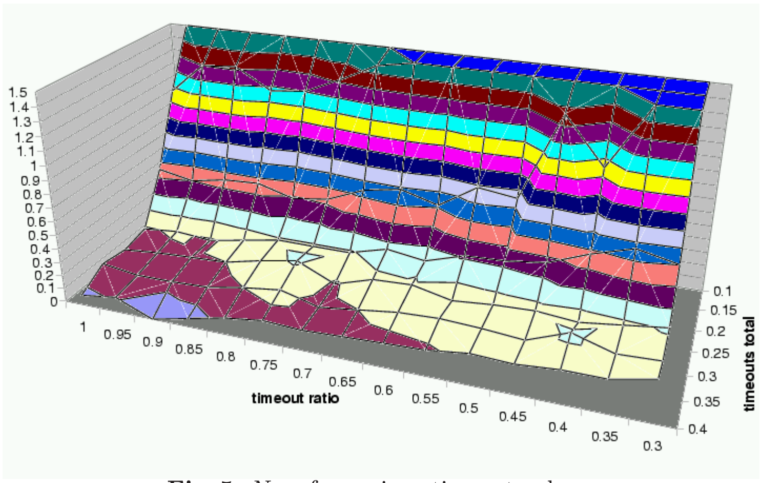

Fig. 5. N_{late} for various timeout values.

effect of the ratio TOS / TOF (x-axis) on the average number of times a present device is wrongly considered to be absent N_{false} (left y-axis) while keeping the sum TOF+TOS (right y-axis) constant. Note that the sum of the timeouts gives an indication about the time a CP needs to determine that a device is absent, as this equals TOF+3·TOS. From the plot we infer that for a fixed ratio of the timeout values, N_{false} rapidly grows when the sums of the timeouts exceeds 0.3 seconds. This effect is not surprising since on shorter timeout periods, a CP will sooner decide that a present device is absent. For a fixed sum of the timeouts, N_{false} slowly decreases on increasing the timeout ratio as the number of (short) timeout periods in which the absence can be wrongly concluded is decreasing. Fig. 6 indicates the bandwidth usage and shows that for a fixed ratio, B grows in this case up to a factor of about 25%. A similar effect can be observed for a fixed total timeout value when the ratio is increased: in case the first timeout period is much longer than the other ones (i.e., TOS/TOF is small), the probability to get a reply on the first probe is relatively large, and there is no need to carry out any retransmissions. If these periods get shorter, this probability is lower, leading to more probes. Using the simulations, the parameters that seem to be adequate for the ping protocol were determined as indicated in the second row of Table 1. Note that in particular, the factors α_{inc} and α_{dec} have changed substantially, as well as the length of the timeout periods. Furthers experiments indicate that with these new parameter values B is basically unchanged, whereas P_{late} is improved significantly, e.g., from 1.772% to 0.718% for the scenario described earlier. More analysis results can be found in [23].

Individual starvation of CPs. The self-adaptive mechanism to control the probe frequency of a device aims at speeding up CPs (i.e., increasing their probing frequency) when other CPs leave the network and at slowing them down when other CPs join. The implicit assumption of this mechanism to work is that all CPs

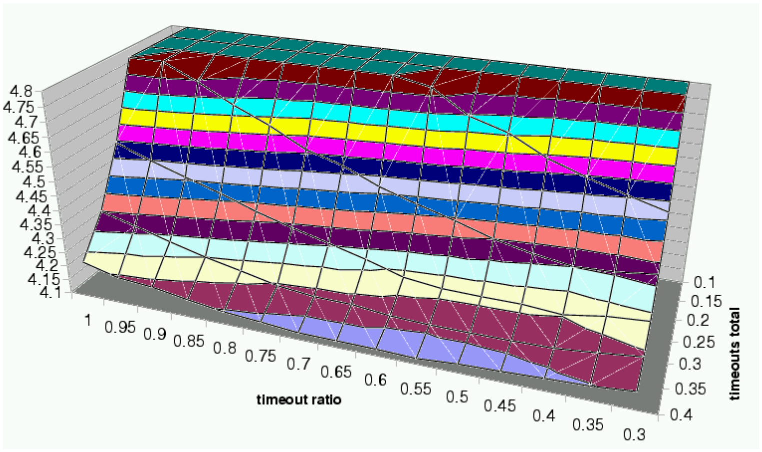

Fig. 6. B for various timeout values.

Table 1. Parameter values prior and after simulations.

	Δ	γ_{max}	γ_{min}	δ_{max}	δ_{min}	α_{inc}	α_{dec}	TOF	TOS
Prior to analyis	4	24	12	30	0.5	$\frac{2}{3}$	2	40	24
After analysis	4	22	14	30	0.5	$\frac{1}{3}$	$\frac{3}{2}$	96	68

are probing at a more or less equal frequency. To obtain an equal spreading of the probing frequency of a device among the probing CPs, slower CPs (that, e.g., just joined) should be able to speed up such that they can match the probing frequency of other CPs. The plot in Fig. 7 shows the spreading of the CPs' probing frequencies in terms of bandwidth usage for a scenario in which four CPs are continuously present, while the other four CPs join at intervals of 50 seconds and leave one by one. The protocol runs according to the parameter values determined in the previous experiment (see above). The individual bandwidth usage of each CP is indicated by a colored curves that are put on top of each other in order to avoid blurring the plots. The lower four curves indicate the bandwidth usage of the static CPs, whereas the upper four curves refer to the CPs that dynamically join and leave the system.

A few remarks are in order. The regular pattern of the CPs joining at regular intervals is clearly recognizable: at each 50 sec there is traffic peak. More importantly, though, is the discrepancy in probing frequencies among the four static CPs: from $t = 250$ on the frequencies of two of these CPs goes towards zero. The protocol mechanism to adapt the probing frequencies also seems not to be able to recover from this problem. The occurrence of the starvation phenomenon can be explained as follows. Suppose several CPs suddenly leave the network in the situation that a slow CP probes at a much lower frequency than another CP. The fast CP detects the absence of the CPs and increases its probing frequency. The slow CP detects this absence much later and the decrease in probeload it

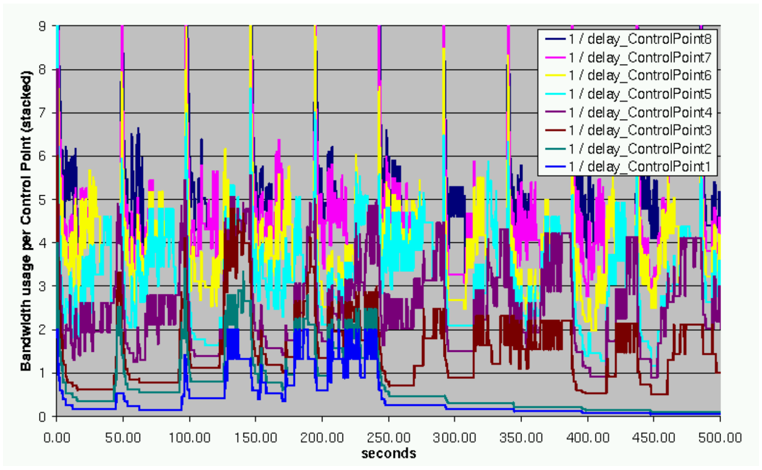

Fig. 7. Bandwidth usage of several CPs.

detects is insufficient to drastically increase its own probing frequency. In the meanwhile the fast CP has speeded up such that the probing frequency of the device is at an appropriate level, and the slow CP does not have any opportunity to increase. In fact, it slows down further. The main cause is that a CP cannot distinguish between the situation in which various other CPs probe a device at a relatively low frequency, and a few (e.g, one) CPs that probe the device at a high frequency.

The starvation problem that was discovered during our analysis was unknown to the designers of the ping protocol, and has caused a major re-design of the protocol. An extended and more detailed description of our experiences on the modeling and analysis of the ping protocol (and its improvements to circumvent the starvation problem) is currently under development [10].

5 Conclusion and Future Perspectives

In this paper, we surveyed the language Modest that allows for the compositional modeling of complex systems and its accompanying tool-support MOTOR. Our framework is particularly suited for analyzing and describing real-time and probabilistic system aspects. This has been exemplified in this paper by modeling and analyzing the ping protocol, a protocol for checking the absence of nodes by various other nodes in a dynamic distributed system [7]. The key aspect of this protocol is an adaptive mechanism to control the probe load of a node. Our analysis has revealed an undesired side-effect of this adaptive mechanism and has provided useful indications on reasonable parameter values. On the basis of our analysis, a significant re-design of the protocol has been made that is described in a forthcoming paper [10].

Recently, some case studies of a rather different nature have been treated with Modest and MOTOR. [12] studies the effect of resource failures in a hard real-time scheduling problem for lacquer production. It assesses the quality of schedules (that are synthesized using a real-time model checker) in terms of timeliness ("what is the probability that the hard deadline is missed?") and resource utilization and studies – as for the ping protocol – the sensitivity wrt. different reliability parameters. [29] presents the modeling and analysis of (part of) the recent European standard for train signaling systems ETCS that is based on mobile communication between the various components. Critical issues such as "what is the probability that a wireless connection can be established within 5 seconds?" are assessed with Motor.

Issues for future work are, among others, applying Modest to more practical case studies and extending Motor with capabilities to analyze timed automata using the real-time model checker UPPAAL [2] and Markov chains using the probabilistic model checker ETMCC [25]. We are currently linking Motor to the in-house conformance test-tool ToRX [4] to enable the on-the-fly automated test generation for real-time (and untimed) systems, and have plans to apply this to the testing of wafer-stepper machines for chip production.

The Motor tool is publicly available from the web-site

`fmt.cs.utwente.nl/tools/motor`

Acknowledgments

The authors thank Jarno Guidi and Lex Heerink from Philips Research Laboratories for pointing out the ping protocol and discussing its details with us. We thank Pedro D'Argenio for his contributions to the Modest semantics and Ed Brinksma for inspiring discussions concerning the language design. The model and results for the ping protocol in this paper are based on the work by Johan Gorter who modeled and analyzed the protocol as part of his master's thesis [23].

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Th. Comp. Sc.*, **126**(2):183–235, 1994.
2. T. Amnell, G. Behrmann, J. Bengtsson, P.R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K.G. Larsen, M.O. Möller, P. Pettersson, C. Weise, W. Yi. Uppaal – Now, next, and future. In: *Modeling and Verification of Parallel Processes*, LNCS 2067:99-124, 2000.
3. C. Baier, F. Ciezinski and M. Groesser. Probmela: a modeling language for communicating probabilistic processes. In: *Int. Conf. on Formal Methods and Models for Codesign*, ACM Press, 2004.
4. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw and L. Heerink. Formal test automation: a simple experiment. In: *Int. Workshop on Testing of Communicating Systems XII*, pp. 179 - 196, Kluwer, 1999.
5. M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

6. G. Berry. Preemption and concurrency. In: R.K. Shyamasundar, ed, *Found. of Software Techn. and Th. Comp. Sc.*, LNCS 761: 72–93, 1993.
7. M. Bodlaender, J. Guidi and L. Heerink. Enhancing discovery with liveness. In: *IEEE Consumer Comm. and Networking Conf.*, IEEE CS Press, 2004.
8. H. Bohnenkamp, T. Courtney, D. Daly, S. Derisavi, H. Hermanns, J.-P. Katoen, V. Lam and W.H. Sanders. On integrating the Möbius and MoDeST modeling tools. *Dependable Systems and Networks*, pp. 671–672, 2003, IEEE CS Press.
9. H. Bohnenkamp, P.R. D'Argenio, H. Hermanns, J.-P. Katoen and J. Klaren. Modest: A compositional modeling formalism for real-time and stochastic systems. 2004 (in preparation).
10. H. Bohnenkamp, J. Gorter, J. Guidi and J.-P. Katoen. A simple and fair protocol to detect node absence in dynamic distributed systems. 2004 (in preparation).
11. H. Bohnenkamp, H. Hermanns, J.-P. Katoen and J. Klaren. The Modest modelling tool and its implementation. In: *Modelling Techniques and Tools for Comp. Perf. Ev.*, LNCS 2794, 2003.
12. H. Bohnenkamp, H. Hermanns, J. Klaren, A. Mader and Y.S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In: *Quantitative Evaluation of Systems*, IEEE CS Press, 2004 (to appear).
13. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. and Comp.*, **163**:172–202, 2001.
14. Special issue on embedded systems. *IEEE Computer*, **33**(9), 2000.
15. M. Bravetti and Gorrieri. The theory of interactive generalized semi-Markov processes. *Th. Comp. Sc.*, **282**(1): 5–32, 2002.
16. P.R. D'Argenio, H. Hermanns, J.-P. Katoen and J. Klaren. Modest: A modelling language for stochastic timed systems. In: *Proc. Alg. and Prob. Methods*, LNCS 2165: 87–104, 2001.
17. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys and G. Tretmans. The bounded retransmission protocol must be on time! In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217: 416–431, 1997.
18. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derasavi, J. Doyle, W.H. Sanders and P. Webster. The Möbius framework and its implementation. *IEEE Tr. on Softw. Eng.*, **28**(10):956–970, 2002.
19. J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A tool box for the verification of LOTOS programs. In *14th IEEE Int. Conf. on Softw. Eng.*, 1992.
20. E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. *SIAM J. Comput.*, **31**(3): 816–837, 2001.
21. H. Garavel and M. Sighireanu. On the introduction of exceptions in E-LOTOS. In: *Formal Description Techniques IX*, pp. 469–484. Kluwer, 1996.
22. P.W. Glynn. A GSMP formalism for discrete event systems. *Proc. of the IEEE*, **77**(1):14–23, 1989.
23. J. Gorter. Modeling and analysis of the liveness UPnP extension. Master's thesis, Univ. of Twente, 2004.
24. H. Hermanns. *Interactive Markov Chains – the Quest for Quantified Quality.* LNCS 2428, 2002.
25. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. A tool for model checking Markov chains. *J. on Software Tools for Technology Transfer*, **4**(2):153–172, 2003.
26. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambr. Univ. Press, 1996.

27. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
28. G.J. Holzmann. *The* SPIN *Model Checker*. Addison-Wesley, 2002.
29. D.N. Jansen, H. Hermanns and Y.S. Usenko. From StoCharts to MODEST: a comparative reliability analysis of train radio communications. 2004 (submitted).
30. M. Jelasity, W. Kowalczyk and M. van Steen. Newscast computing. Tech. Rep. IR-CS-006, Vrije Univ. Amsterdam, 2003.
31. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Th. Comp. Sc.*, **282**:101–150, 2002.
32. E.A. Lee. Embedded software. In: M. Zelkowitz, editor, *Advances in Computers*, vol. **56**, Academic Press, 2002.
33. J.F. Meyer, A. Movaghar and W.H. Sanders. Stochastic activity networks: structure, behavior and application. In: *Int. Workshop on Timed Petri Nets*, pp. 106–115, 1985.
34. G.D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, 1981.
35. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
36. M. Raynal and F. Tronel. Group membership failure detection: a simple protocol and its probabilistic analysis. *Distrib. Syst. Engng*, **6**: 95–102, 1999.
37. R. van Renesse, Y. Minsky and M. Hayden. A gossip-style failure detection service In: *IFIP Conf. on Distributed Systems, Platforms, and Open Distributed Processing*, pp. 55–70, 1998.
38. W.-P. de Roever, F.S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Univ. Press, 2001.
39. W.H. Sanders and J.F. Meyer. Stochastic activity networks: formal definitions and concepts. In: *Formal Methods for Performance Evaluation*, LNCS 2090: 315–344, 2001.
40. F.B. Schneider, B. Bloom and K. Marzullo. Putting time into proof outlines. In: *REX Workshop on Real-Time:Theory in Practice*, LNCS 600: 618-639, 1991.
41. R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic J. of Computing*, **2**(2): 250–273, 1995.