Probabilistic Analysis of Embedded Systems

Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen

Abstract This chapter provides a gentle introduction into compositional modeling of systems that involve nontrivial real-time and probabilistic aspects. It revolves around the language MODEST, a Modelling and Description language for Stochastic and Timed systems. The language supports the compositional description of reactive systems while covering both functional and non-functional system aspects such as quantified component failure rates and hard and soft real-time. A running example illustrates the language constructs. Afterwards, this example is used to highlight different avenues to analyse such models that have been implemented in a tool suite. Among them, we find probabilistic timed model checking as well as discrete event simulation.

1 Introduction

There is a growing awareness among embedded software designers that the classical computer science approach—to abstract from physical aspects—is too limited and too restricted for contemporary and upcoming design challenges [1, 13]. Instead, abstractions of software that leave out "non-functional" aspects such as cost, efficiency, and robustness need to be adapted to current needs.

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

Arnd Hartmanns
Saarland University – Computer Science, e-mail: arnd@cs.uni-saarland.de

Holger Hermanns
Saarland University – Computer Science, e-mail: hermanns@cs.uni-saarland.de

Joost-Pieter Katoen
University of Twente, RWTH Aachen University – Computer Science, e-mail: katoen@cs.rwth-aachen.de

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 its—mostly physical—environment via sensors and actuators. Software in embedded systems does not typically 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 modelling 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 UML (Unified Modeling Language), and in the development of a whole range of more rigorous formalisms based on, for example, stochastic process algebras, or appropriate extensions of automata such as timed automata [2] and probabilistic automata [16]. Light-weight notations are typically closer to engineering techniques, but lack a formal semantics; rigorous formalisms do have such a formal semantics, but their learning curve is typically too steep from a practitioner's perspective and they mostly have a restricted expressiveness.

This paper surveys MODEST [3], a description language that has a rigid formal basis (i.e., semantics) and incorporates several ingredients from light-weight notations such as exception handling, modularization, atomic assignments, iteration, and simple data types. The paper also illustrates advanced tool support, all by means of a running example.

MODEST is a *compositional* specification formalism: the description of complex behaviour is obtained by combining the descriptions of simpler components. This provides an elegant way to specify concurrent computations, inherited from process algebra. 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 [2, 4], probabilistic variants thereof [5], and stochastic processes such as Markov chains and (continuous-time and generalised) Markov decision processes [8, 10, 15, 16].

MODEST takes 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 UML. Analysis takes place by extracting simpler models from MODEST specifications that are tailored to the specific property of interest. For instance, to check 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 [11] or CADP [7]. For probabilisitic timed models, we implement a translational model checking approach [9], using the PRISM tool as backend. On the other hand, to carry out an evaluation of the stochastic process underlying a MODEST specification, we support discrete-event simulation.

2 Modelling with MODEST

In this section, we will introduce the MODEST language syntax and its semantic basis by modelling a simple communication scenario. We will introduce the language features and constructs step-by-step, starting with a very basic functional model which we then extend to include timed, probabilistic and stochastic behaviour. While the focus of this section is on the language syntax and the types of behaviours than can be modelled, we will also give brief insights into the underlying semantics where useful for a deeper understanding.

2.1 Syntax and Semantics Basics

The scenario we are going to model in MODEST is a simple communication setting where a sender continuously tries to send messages to some receiver over an unreliable channel that may lose, but not reorder or create messages. We will refine our models in the following sections; let us start with a very basic functional description of sender, receiver and channel for now.

In the MODEST language, being inspired by classic process-algebraic languages such as CCS, CSP and LOTOS, everything is a process. Processes can perform actions, and in this way transition into a different process. One of the most basic processes in MODEST is `tau`, which can perform a single action named *tau*, transitioning into a process than cannot do anything, which we denote by ✓ if it expresses a situation corresponding to "successful termination". Processes can also be given names, allowing them to be reused in other places.

Our first example, the `Receiver` process shown in Figure 1, uses the sequential composition construct `;` to combine several basic processes (that each just perform an action) into a sequence of processes that each still perform a single action, but then transition into the next one. While the semantics of the `;` construct is intuitively clear, a formal definition of the semantics of all constructs of the MODEST language is necessary for formal verification. The result of applying this semantics [3] to a given process is an automaton whose states correspond to MODEST processes, and whose edges are labelled with actions and represent the transitions between processes. For now, the resulting automaton is just a labelled transition system (LTS), the simplest submodel supported by the MODEST language.

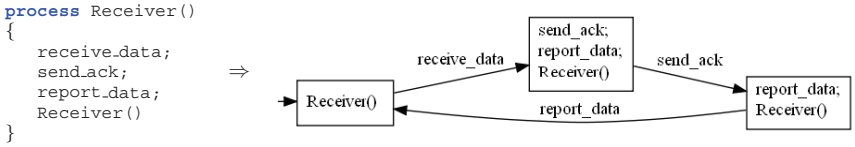

Fig. 1 The most basic model of a receiver

Figure 1 also shows the LTS corresponding to the `Receiver` process. The behaviour of the receiver is thus to—upon receiving some data from the communication channel—first send an acknowledgment back to the sender, then report the arrival of the message (presumably to some upper network layer), and finally start over again and wait for new data. In the following, we may sometimes omit location labels and certain parts of the edge labels for clarity; however, keep in mind that a location always corresponds to a MODEST process.

2.2 Nondeterminism

An important feature of many modelling formalisms, including MODEST, is to allow nondeterministic specifications. A nondeterministic choice is a choice between two different courses of action without any information about the likelihood of one of them or the circumstances that may lead to it. As such, it can be used to model the complete absence of knowledge about the actual behaviour of a system; it can be used to intentionally leave an implementation choice in a specification; and it allows an open model to react on stimulus from a yet unknown environment.

We use a nondeterministic choice for the latter purpose in our first model of the sender: To allow guaranteed delivery, the sender waits for an acknowledgment from the receiver that the message it just sent has arrived before moving on to the next one. Since the communication channels are lossy, the actual data or the receiver's acknowledgment may be lost, so the sender may have to repeat its transmission. We use a nondeterministic choice between receiving an acknowledgment and detecting that a message has been lost, which will eventually be resolved by the actual behaviour of the environment.

The MODEST model for the sender and the corresponding LTS is shown in Figure 2. The **`alt`** construct is used to specify the nondeterministic choice between receiving an acknowledgment and detecting message loss, the latter of which is encapsulated in a dedicated process `Timeout` that we specify as

```
alt { :: timeout_send  :: timeout_ack }
```

for this first model. Note that instead of calling `Sender()` after the receipt of an acknowledgment or timeout to start over again, we chose MODEST's loop construct, **`do`**, which causes some process to be repeated ad infinitum or until it issues the special **`break`** action.

```
process Sender()
{
   do {
      :: send_data;
         alt {
            :: receive_ack
            :: Timeout()
         }
   }
}
```

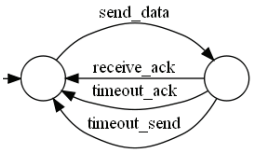

Fig. 2 The most basic model of a sender

```
process Channel()
{
   receive;
   alt {
      :: send
      :: drop
   };
   Channel()
}
```

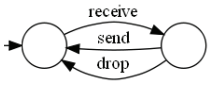

Fig. 3 A simple lossy communication channel

The only component of our communication scenario that still needs to be modelled is the channel. Its model, shown in Figure 3, again uses a nondeterministic choice. However, this time, it represents an "absence of knowledge" case: We do not know anything about the channel except for the fact that it *may* lose messages, so after receiving a message on one end, we just nondeterministically allow both possibilities—successful propagation of the message to the other end (`send`) or message loss (`drop`).

2.3 Processes Running in Parallel

Although we now have MODEST processes that represent all components of our communication scenario—sender, receiver and channel—we still need to obtain a single model for the whole system that also includes the interactions between the different components. Since these components usually represent distinct physical entities that mostly run independently from each other, possibly with different processing speeds, their composition is best represented by letting them run in parallel without further restrictions, allowing their actions to occur interleaved in any order. Only if the components actively interact with each other may we need to model a synchronisation between them.

In MODEST, this kind of parallel composition is implemented by the **`par`** construct: The parallel composition of n processes P_1 to P_n, **`par`** $\{ :: P_1 \ \ldots \ :: P_n \}$, allows the processes to perform their actions in any order, unless an action is shared by at least two processes. In that case, in order to perform such a shared action, all the processes that contain the action have to perform it at the same time, as a sin-

```
par {
    :: Sender()
    :: relabel {receive, send, drop} by {send_data, receive_data, timeout_send}
       Channel() // Channel from sender to receiver
    :: relabel {receive, send, drop} by {send_ack, receive_ack, timeout_ack}
       Channel() // Channel from receiver to sender
    :: Receiver()
}
```

Fig. 4 MODEST code for the parallel composition of sender, receiver and communication channels

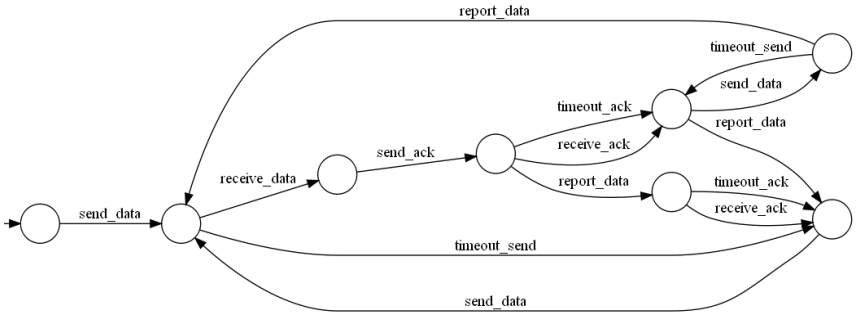

Fig. 5 LTS of the parallel composition of sender, receiver and communication channels

gle step. This synchronisation on shared actions allows us to model communication between processes.

Figure 4 shows the parallel composition of the sender, one instance of the `Channel` process to model the channel from sender to receiver, another instance to model the channel from receiver to sender, and the receiver itself. We use the **`relabel`** construct to rename actions in such a way that sender and receiver synchronise with the two channels as intended.

In our example, the LTS corresponding to the parallel composition (Figure 5) is still relatively small, but this is only because the individual processes synchronise very often. In theory, the size of the LTS of a parallel composition is only bounded by the product of the numbers of states of the individual processes, and may thus grow large very quickly.

2.4 Data Exchange

The basic model of the communication scenario we developed so far has one serious problem: The receiver will report every receipt of a message from the channel, even if it is just a retransmission after a lost acknowledgment. This is clearly undesirable behaviour, but in order to fix it, we need a way to distinguish different messages. A classic solution is that of the *Alternating Bit Protocol*: If we can guarantee that there is at most one message in transit at any time, it suffices to include a single bit in every message that is flipped between subsequent messages. The receiver stores the

```
bool channel_bit;

process Sender()
{
   bool bit;

   do {
      :: send_data {= channel_bit = bit =};
         alt {
            :: receive_ack {= bit = !bit =}
            :: Timeout()
         }
   }
}

process Receiver(bool last_bit)
{
   bool bit;

   receive_data {= bit = channel_bit =};
   urgent send_ack;
   alt {
      :: when(bit != last_bit) report_data
      :: when(bit == last_bit) tau
   };
   Receiver(bit)
}
```

Fig. 6 Transmission of an alternating bit between sender and receiver

value from the last reported message, and if a new message arrives with the same value, it clearly is a retransmission.

MODEST supports data in the form of global and process-local variables, including parameters to processes, which can be of Boolean, integer, bounded integer or real type, arrays thereof, or user-defined composite types. In order to transmit a single bit, a Boolean variable is sufficient. The necessary modifications to the model are shown in Figure 6: A global variable `channel_bit` is added that represents the bit of the message that is currently being transmitted, and sender and receiver get local variables or parameters to keep track of the current and previous bits.

Aside from the variable declarations, we see two new language features in the modified model, namely assignments and the **when** construct, or *guard*. Assignments are associated with an action and enclosed in brackets ({=. . . =}). They are executed atomically when the action is performed; in particular, the textual order of the assignments in MODEST code inside a {=. . . =} block does not matter. The variables can then be used to put constraints on when an action can actually be performed by using the **when** construct.

2.5 Time

Up to this point, the detection of message loss was implemented by the process

```
alt { :: timeout_send  :: timeout_ack }
```

```
const int TD; // maximum channel transmission delay
const int TS; // sender timeout

process Timeout()
{
   clock c;

   when(c >= TS) urgent(c >= TS) tau // timeout: retransmit
}

process Channel()
{
   clock c;

   receive {= c = 0 =};
   alt {
      :: urgent(c >= TD) send
      :: urgent tau // silently drop the message
   };
   Channel()
}
```

Fig. 7 Adding a transmission delay and realistic timeout detection

where the two `timeout_*` actions just synchronised with the `drop` actions that the channels performed when they lost a message. This is, of course, a significant abstraction from reality where message lost is usually detected by the passage of a certain amount of time, after which a *timeout* occurs. Fortunately, time is supported in MODEST as well, in a way that is almost identical to how time is modelled in Timed Automata (TA, see Chapters 2 and 3), so we can now make our model more realistic w.r.t. timeouts.

Just like TA, MODEST has clock variables, which can be also be used in expressions in guards and *deadlines*. Deadlines (or *urgency constraints*, written in MODEST as **`urgent`**`(`d`)` where d is an expression of Boolean type) are used to constrain the passage of time; they are the MODEST analogue to invariants in TA. In contrast to invariants, however, they are not associated with states, but with the actions possible in a process. If the deadline of any action possible in a process becomes true (we say that the action becomes *urgent*), time cannot pass any more and some edge has to be taken.

Figure 7 shows how to use these new constructs to put realistic timeouts into our model. The transmission of a message through a channel now takes up to `TD` time units (which is achieved by resetting clock `c` to zero and then prefixing the channel's `send` action with **`urgent`**`(c >= TD)`), and the `Timeout` process is modified such that it actually waits some time—precisely `TS` time units—before terminating and thereby causing a retransmission, leaving the bit unchanged. The corresponding automaton is shown in Figure 8—note that guards, deadlines and assignments are stored symbolically on the transitions.

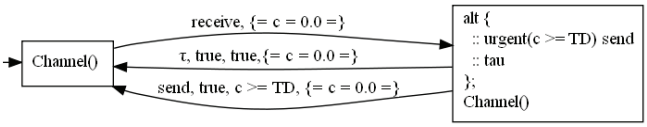

Fig. 8 Automaton for the channel with transmission delay

```
process Channel()
{
   clock c;

   receive palt {
      :98: urgent(c >= TD) send
      : 2: urgent tau // silently drop the message
   };
   Channel()
}
```

Fig. 9 A timed probabilistic lossy channel

2.6 Probabilistic Choices

Although our model of the communication scenario has become significantly more realistic through the addition of "real" timeouts, there still is one more problem with the current channel model: The decision whether or not to lose a message is currently a nondeterministic one. While this is a good model for complete absence of information about the frequency of message losses, one possible resolution of this choice is to always drop the message, i.e., a completely non-functional channel. In order to avoid this situation, we either need so-called fairness assumptions which may enforce that eventually some message is not lost, or we just have to put information about the frequentness of message loss into the model.

Let us pursue the latter option for this example. Experiments may have shown that for a typical implementation of this kind of channel, a single message is lost with a probability of 2%, independent of any other conditions. This *probabilistic choice* between two options is available in MODEST via the `palt` construct, as shown in Figure 9: After receiving a message at one end of the channel, it is now lost with probability $\frac{2}{98+2} = 2\%$, and it successfully arrives at the other end (possibly with some delay) with probability 98%. The probabilities used in the `palt` construct are actually *weights*, so we could equivalently have written 49 and 1 instead of 98 and 2, and they are not restricted to constant values as in the example, but can be arbitrary expressions.

For the previous examples, the semantics of our model was an automaton with variables and edges that consisted of an action, a guard, a deadline, and assignments. When we add the `palt` construct, however, the model becomes slightly more involved because we need to represent the probabilistic branching in the automaton. As shown in Figure 10, edges can now target multiple locations, with a probability given for each branch. (Formally, the edges now relate a source location with a probability distribution over target locations and assignments.) This is also why

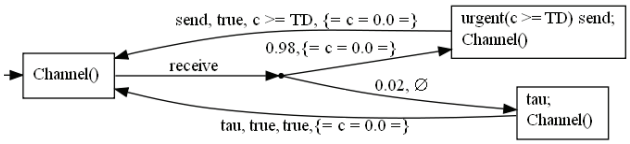

Fig. 10 Automaton for the channel with transmission delay and probabilities

the `palt` construct has to be prefixed with an action—it needs an existing edge to which it can add probabilistic branching.

2.7 Random Timing

The model built so far for our communication scenario could already be considered complete; however, let us add one more refinement by explicitly modelling the arrival of new messages for the sender. We will add a queue of messages that periodically grows and out of which the `Sender` process takes a message as soon as the transmission of the previous one is successfully completed.

The time between the arrivals of "customers", in our case messages, to a queue is usually assumed to be exponentially distributed with some rate parameter λ. To model these delays, we can take advantage of the possibility to draw samples from random variables distributed according to a number of predefined distributions in MODEST: for example, the statement

```
{= delay = Exponential(3.14) =},
```

assigns a value to the variable `delay` (which is of type real) that is randomly chosen according to the exponential distribution with rate 3.14. We can then use this variable in guards and urgency constraints to achieve a delay that is of exponentially distributed length, as long as we resample before using it again:

```
urgent(c >= delay) when(c >= delay) ...
```

This now allows us to implement the queue of messages as shown in Figure 11. We have also modified the sender by adding a `get_data` action that synchronises with the `Queue` process, taking out one message.

2.8 More Syntax

The full communication scenario model (Figure 11) concludes our tour of the MODEST language. We have seen that MODEST can be used to build models of real-time systems with probabilistic or stochastic behaviour, including how its process-algebraic nature can be used to specify a complex system in terms of its natural components, which themselves remain small and easy to understand.

```
action get_data;
action receive_data, send_ack, report_data;
action send_data, receive_ack;
action receive, send;

const int TD; // maximum channel transmission delay
const int TS; // sender timeout
const real AR; // data arrival rate

bool channel_bit;

process Timeout() { ... }

process Sender(bool bit)
{
   get_data;
   do {
      :: urgent send_data {= channel_bit = bit =};
         alt {
            :: receive_ack; urgent break
            :: Timeout()
         }
   };
   Sender(!bit)
}

process Receiver(bool last_bit) { ... }

process Channel() { ... }

process Queue()
{
   clock c;
   int items;
   real delay;

   do {
      :: urgent(c >= delay) when(c >= delay)
         {= items += 1, delay = Exponential(AR), c = 0 =}
      :: when(items > 0) urgent(items > 0)
         get_data; urgent {= items -= 1 =}
   }
}

par {
   :: Queue()
   :: Sender(false)
   :: relabel {receive, send} by {send_data, receive_data}
      Channel() // Channel from sender to receiver
   :: relabel {receive, send} by {send_ack, receive_ack}
      Channel() // Channel from receiver to sender
   :: Receiver(true)
}
```

Fig. 11 The full model

MODEST is an expressive language, and although we essentially covered the types of behaviours that are expressible in MODEST, we were not able to present all of its syntactic features. Most notably, it also support exceptions as known from modern programming languages that can be thrown at some point in a model and be caught at another, but also several useful shorthands that, for example, allow the use of invariants as known from timed automata instead of deadlines.

Submodel	Probability distributions?	Clocks/Time?	Nondeterminism?
STA	arbitrary	arbitrary	yes
GSMP	arbitrary	arbitrary	no
PTA	finite	integer bounds	yes
TA	none	integer bounds	yes
PA/MDP	finite	no	yes
LTS	none	no	yes
CTMDP	exponential + finite	exponential delays	yes
CTMC	exponential + finite	exponential delays	no
DTMC	finite	no	no

Table 1 MODEST submodels

3 Analysing MODEST Models

The ultimate goal of modelling any kind of system is to be able to analyse the model, and in particular verify the presence or absence of certain good or bad properties. Due to the enormous expressiveness of MODEST, no currently known analysis technique can be used for arbitrary MODEST models. However, certain submodels of MODEST (Table 1) can easily be identified, e.g. by the absence of certain language constructs, and efficient analysis techniques targeted to several of these submodels exist:

Two very general submodels underlying MODEST are probabilistic timed automata (PTA, [5]) and generalized semi-Markov processes (GSMP, [8]). The STA corresponding to a MODEST model is a PTA if only probability distribution with finite support set occur. The complete model of our running example from the previous section is thus not a PTA because it uses the exponential distribution, but the one developed up to Section 2.6 is. We will show how to analyse this model in the first part of this section. On the other hand, regardless of the probability distribution appearing, a MODEST model corresponds to a GSMP, provided it is fully deterministic. Unfortunately, this condition is difficult to assure — in particular, the parallel composition is not closed under determinism: fully deterministic processes running in parallel may behave nondeterministically.

3.1 Model-Checking PTA Models

For the formal verification of MODEST models that correspond to PTA, a set of properties to be checked has to be defined. As PTA combine probabilistic and real-time behaviour, we can refer to both probabilities and time in these properties. Some classes of probabilistic timed properties that can efficiently be verified are

- *probabilistic reachability properties*: “What is the probability of ever reaching an error state?”,

- *probabilistic time-bounded reachability properties*: "What is the probability of reaching an error state within *n* time units?", and
- *expected-time reachability properties*: "What is the expected time until an error state is reached?".

Because PTA can contain nondeterminism, the answers to all of these properties depend on how the nondeterminism is resolved. All of them thus exist in a *maximum* and *minimum* variant: If asked for the maximum (minimum) probability of reaching an error state, all possible ways to resolve nondeterministic choices are considered and the highest (lowest) probability is returned.

Let us now analyse the model of the communication scenario developed up to Section 2.6 in terms of correctness and performance. The most basic correctness criterion for such a communication protocol is that the probability of eventually succeeding, which we may, for example, detect by the receipt of an acknowledgment, is 1. This can be specified inside the model as

```
property success = P(<> did(receive_ack)) >= 1.0;
```

For `TD` = 1 and `TS` = 4, this property is indeed satisfied. Now that we are confident that the protocol used is correct, we can study performance aspects. A requirement may for example be that, in the worst case, the probability of completing a transmission within 7 time units is close to 100 %. For the same values for `TD` and `TS`, we will see that it is actually 99.843 % using

```
property Pmin(<> did(receive_ack) && time <= 7.0);
```

Lastly, we can also find out what the actual expected time until successful transmission is. Using the following properties—

```
property success_time_min = Tmin(did(receive_ack));
property success_time_max = Tmax(did(receive_ack));
```

—we see that it lies between 1.649 and 2.165 time units, depending on how the nondeterministic choice of the actual transmission delay in the channel is resolved.

3.2 Discrete-Event Simulation for GSMP Models

Models using probability distributions with infinite support, such as the exponential distribution, can be analysed using discrete-event simulation. Since discrete event simulation relies on the execution and evaluation of large batches of *concrete* traces of a model, this model either needs to be deterministic (i.e., a GSMP), or the nondeterminism has to be resolved in some way specified by the user in order to obtain a GSMP.

There are many different methods to resolve nondeterminism, and the particular method employed may skew the analysis results in unexpected and sometimes counterintuitive ways. For example, two distinct possibilities to resolve nondeterminism over time – such as for the transmission delay in our running example – are to always choose the earliest or the latest possible point in time. For our example,

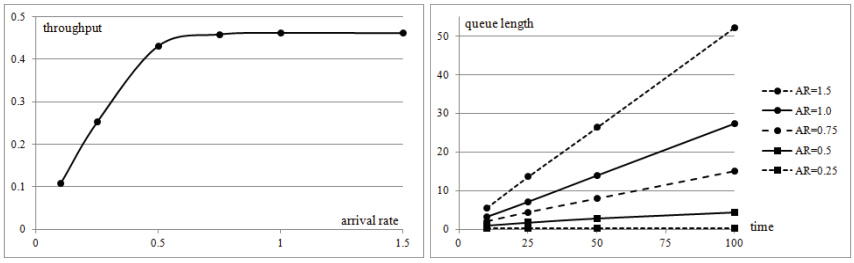

Fig. 12 Simulation results for the communication scenario

the results we obtain this way for the expected time to success do coincide with the actual lowest and highest values, but it is easy to construct a model where, for example, the inverse is the case. Understanding that the method to resolve nondeterminism can influence the results and taking this into account when interpreting the results is thus crucial.

When analysing the final model of our communication scenario via discrete-event simulation, we can still check the properties presented for the PTA model above, but also new measures such as the actual throughput of the system in terms of successfully transmitted messages per time unit and the average length of the queue over time. The former can be achieved by simply counting the number of times that the `report_data` action is performed via MODEST's support for transition rewards:

```
increment count for {report_data} by {1} Receiver(true)
```

At the end of a simulation run, we then observe the value of `count/time`. To measure the average queue length, we need to use a rate reward that grows (linearly) over time at the speed of the current queue length. We can specify this in MODEST by introducing a new variable `items_reward` of type real and setting its derivative to be the queue length:

```
der(items_reward) = items;
```

Again, we let the simulator observe the value of `items_reward/time` at the end of each simulation run.

Discrete-event simulation is usually used to perform a large number of simulation runs and then compute a statistical evaluation of the collected results. To obtain numbers for the measures introduced above, we collect 1000 random traces for different model time lengths with uniformly random resolution of nondeterminism, but a deterministic transmission delay of 1 time unit and $\mathtt{TS} = 4$; the results are plotted in Figure 12. We see that the throughput of the system is limited to about 0.46 messages per time unit due to the communication delays and the lengths of the timeouts in case a message is lost; as expected, the queue length appears to grow without bounds for arrival rates close to or larger than that value; slowly so for $\mathtt{AR} = 0.5$, but already significantly for $\mathtt{AR} = 0.75$.

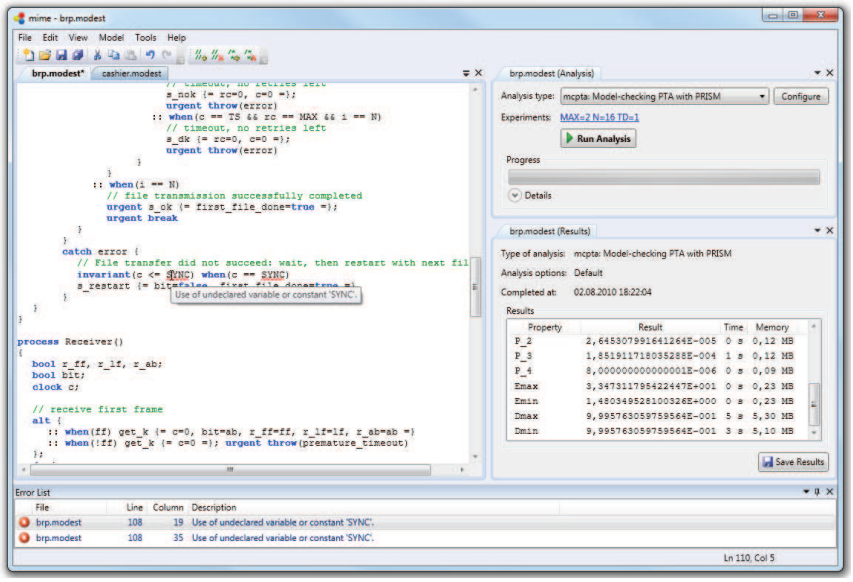

Fig. 13 mime, the integrated modelling environment for MODEST

3.3 Tool Support

The analysis of MODEST models is today supported by two sets of tools: The newly developed *Modest Toolset* and the original *Modest Tool Environment*, MOTOR.

3.3.1 The Modest Toolset

The MODEST Toolset, developed at Saarland University, currently consists of three tools: *mcpta*, which allows model-checking MODEST models of PTA [9]; *modes*, a discrete-event simulator for MODEST, and *mime*, an integrated modelling environment that combines a MODEST editor with syntax and error highlighting with direct access to the model analysis capabilities of mcpta and modes (Figure 13). The MODEST Toolset is cross-platform and can be downloaded at

`www.modestchecker.net`

mcpta transforms MODEST models corresponding to PTA into probabilistic, but untimed models and hands these over to the PRISM probabilistic model-checker [14] for analysis. This process is fully automated, and all classes of properties introduced in Section 3.1 are supported.

modes' focus is on discrete-event simulation with a sound treatment of nondeterminism. Its default behaviour is therefore to reject nondeterministic models, but it allows the user to override this behaviour by explicitly choosing one out of a set

of predefined resolution methods. Additionally, modes can be configured to detect and ignore certain kinds of *spurious* nondeterminism, i.e. choices that do not actually influence the final result. When this option is used, modes can analyse (certain) nondeterministic models while the user can rest assured that the values obtained are unaffected by any particular resolution of nondeterminism.

3.3.2 MoTor and Möbius

MOTOR [12] is the original set of tools designed to interface MODEST with different existing analysis backends such as CADP [7] for functional verification and MÖBIUS [6] for discrete-event simulation. Today, the MÖBIUS backend is mostly used for high-performance and distributed simulation of MODEST models; see the following chapter for an extensive case study. In contrast to modes, nondeterministic choices over different actions are always resolved in a uniformly probabilistic way, while a *maximal progress* semantics is used for nondeterministic delays (that is, as soon as an action is possible, it is taken, even when the model would allow more time to pass).

MOTOR is available from the University of Twente at

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

3.3.3 Analysing other Submodels

While not yet supported by the tools introduced above, several other submodels of MODEST are easy to analyse with well-established model-checking tools once the MODEST code is translated into the respective tool's formalism: For example, the timed automata subset could be analysed using UPPAAL, while the one corresponding to continuous-time Markov chains (CTMC) could also be translated to PRISM. Since Markov decision processes (MDP) and discrete-time Markov chains (DTMC) are special cases of PTA, mcpta can already be used in combination with PRISM to analyse these submodels.

4 Summary

Summary/conclusion **`HH: TODO HH:`**.

References

1. IEEE Computer, special issue on embedded systems, 2000.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

3. Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.
4. Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.
5. Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the IEEE-1394 root contention protocol with KRONOS and PRISM. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.
6. Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Software Eng.*, 28(10):956–969, 2002.
7. Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.
8. Peter W. Glynn. A GSMP formalism for discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 14–23. 1989.
9. Arnd Hartmanns and Holger Hermanns. A Modest approach to checking probabilistic timed automata. In *QEST '09: Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, pages 187–196, Washington, DC, USA, 2009. IEEE Computer Society.
10. Holger Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.
11. Gerard J. Holzmann. Software analysis and model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.
12. Joost-Pieter Katoen, Henrik C. Bohnenkamp, Ric Klaren, and Holger Hermanns. Embedded software analysis with MOTOR. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 268–294. Springer, 2004.
13. Edward A. Lee. Embedded software. *Advances in Computers*, 56:56–97, 2002.
14. David Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
15. M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Inc., New York, 1994. A Wiley-Interscience Publication.
16. Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.