Modelling Stochastic Timed Systems

Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, Ric Klaren
Formal Methods and Tools Group, Faculty of Computer Science
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: {dargenio, hermanns, katoen, klaren}@cs.utwente.nl

Abstract— **Real-time, performance and reliability aspects are of vital importance in the entire system design trajectory. Therefore, modelling techniques are needed that cover quantitative system aspects. This paper presents MoDeST, a modelling language that allows us to specify soft real-time constraints (i.e., stochastic timing) as well as hard real-time constraints. MoDeST combines conventional programming constructs – such as iteration, alternatives, atomic statements, and exception handling – with means to describe complex systems in a compositional manner. The language is influenced by popular and user-friendly specification languages, and deals with compositionality in a light-weight process-algebra style. In summary, MoDeST (i) covers a very broad spectrum of modelling concepts, (ii) possesses a rigid, process-algebra style semantics, and (iii) yet provides modern and flexible specification constructs.**

Keywords— **hard real-time, soft real-time, probabilistic behaviour, specification language, process algebra**

I. Introduction

System design is primarily focussed on functional aspects. Non-functional aspects such as reliability and performance typically play a role – if at all – in the final stages of the design trajectory. To overcome this problem, sometimes identified as the insularity problem of performance engineering [14], [11], it has been widely recognised that quantitative system aspects should be considered during the entire system design trajectory. Although a complete insight in the quantitative aspects might not be present at each design stage, even with partial information (or rough estimates) design alternatives may be rejected early due to unsatisfactory performance or dependability characteristics. For this purpose, modelling techniques used by system engineers or those that provide an easy migration path for users need to be adapted to take quantitative system aspects into account.

This has resulted in extensions of light-weight formal notations such as SDL and UML on the one hand, and the development of a whole range of more rigorous formalisms based on e.g., stochastic process algebras, or appropriate extensions of labelled transition systems. Light-weight notations are typically closer to engineering techniques, but mostly lack a formal semantics; rigorous formalisms do have such formal semantics, but their learning curve is typically too steep from a practitioner's perspective. In this paper, we propose a description language that is intended to have a rigid formal basis (i.e., semantics) and incorporates several ingredients from light-weight notations such as exception handling[1], modularisation, atomic statements, iteration, and simple data types. The semantics enables formal reasoning and provides a solid basis for the development of tool support, whereas the light-weight ingredients are intended to pave the migration path towards engineers.

The modelling language presented in this paper, called MoDeST (a MOdelling and DEscription language for Stochastic and Timed systems) contains the following key features:

- light-weight control structures such as iteration, and exception handling
- simple data types that can be user-defined using modularisation (packages)
- composition and abstraction mechanisms to structure specifications
- atomic statements to control the granularity of transitions
- non-deterministic and probabilistic alternatives
- non-deterministic and probabilistic timing (soft and hard real-time constraints).

Organisation of the paper. Section II discusses the main rationales behind the development of MoDeST. Section III introduces the language ingredients of MoDeST in an incremental way. Section IV reviews the syntax and semantics. Section V briefly addresses the current status of our tool support for MoDeST, while Section VI concludes the paper. The paper focuses on behavioural aspects and omits considerations

Supported by the STW-PROGRESS project TES-4999, "HaaST: Verification of Hard and Softly Timed Systems", and the NWO project 612.069.001.

[1]Exception handling in specification languages has received scant attention. Notable exceptions are Enhanced-LOTOS [12] and Esterel [3].

on data manipulation.

II. Design rationales

Important rationales behind the development of the description language, called MoDeST (Modeling and Description language for Stochastic Timed systems), are:

- *Orthogonality.* The language has been set up in an orthogonal way such that timing and probabilistic aspects can easily be added to (or omitted from) a specification if these aspects are of (no) relevance.
- *Usability.* Syntax and language constructs have been designed to be close to other commonly used languages. The syntax resembles that of the programming language C and the modelling language Promela [17]. Data modularisation concepts and exception handling mechanisms have been adopted from modern object-oriented programming languages such as Java [13]. Process algebraic constructs have been strongly influenced by FSP (Finite State Processes [19]) a simple, elegant calculus that is aimed at educational purposes.
- *Practical considerations.* The design of the language and the development of accompanying prototype tool-support have taken place hand-in-hand. Considerations about the tool handling of language constructs have been a driving force behind the language development.
- *Expressiveness.* We have identified a handful of semantic concepts which are well-established in the context of computer-aided verification and modelling formalisms for stochastic discrete event systems:

(1) Action nondeterminism is often used in concurrent system design to leave parts of the description under-specified, and is an appropriate means to reflect that the order of events in concurrent executions is out of the control of a modeller.

(2) Probabilistic branching is a way to include quantitative information about the likelihood of choice alternatives. This is especially useful to model randomized distributed algorithms, but also suitable to represent scheduling strategies, quantify data dependencies etc. on an abstract level.

(3) Clocks are a means to represent real time and to specify the dynamics of a model in relation to a certain time or time interval, represented by a specific value of a clock.

(4) Delay nondeterminism allows one to leave the precise timing of events unspecified. In many cases, the system dynamics depends on events taking place in some time interval (e.g., prior to a time-out) where it is left unspecified when in the interval the event will occur.

(5) Random variables are often used to give quantitative information about the likelihood of a certain event to happen after or within a certain time interval.

While (1) and (2) affect the dynamics of a model via the (discrete) set of next events, (4) and (5) are means to affect the model dynamics by the (continuous) elapse of time. Thus, (1) and (4) describe two distinct types of nondeterminism, while (2) and (5) represent distinct types of probabilistic behaviour. We believe that each of these concepts is indispensable if striving for an integrated consideration of quantitative system aspects during the entire system design trajectory. However, we are not aware of any other formalism, model, or tool that is powerful enough to cover the complete spectrum spanned by this classification. Some approaches however come close, among them [22], [4], [1], [6], [20].

III. A gentle language primer

This section introduces the core language features of MoDeST by specifying a real-time cashier. This is done in an incremental manner starting from an untimed, non-probabilistic description.

A. Functional behaviour

```
process Cashier() {
    do{:: get_prod ; alt {
                      :: cash
                      :: set_price ;
                         cash }
      }
}
```

The system is informally described as follows. In a supermarket customers arrive at the cashing point and queue in order to pay their selected products. The customers provide their products on a conveyor belt and the cashier takes the products one-by-one from the belt (this is modelled by action *get_prod*). The product is either cashed (action *cash*), or in case there is no price tag, the cashier calls for assistance to establish the price (action *set_price*) after which cashing takes place (action *cash*). This behaviour is described by the above process, where ; denotes sequential execution and :: is used as a separator for the different alternatives of the choice construct alt . This construct is a way to model action nondeterminism. The cashier repeats his (or her) behaviour (indicated by do{:: ...} which is executed repeatedly, unless a break occurs).

B. Probabilistic branching

In case more information is available about the likelihood with which a customer delivers a product without price tag, the nondeterministic choice may be replaced by a probabilistic choice. This yields the process below, where weights (in the form of positive reals) are used to determine the likelihood with which a certain alternative should be chosen. Here, price information is available with probability 0.98 and the price tag is absent with probability 0.02. In the terminology of Section II, palt is a means to incorporate probabilistic branching. Each probabilistic choice-construct is required to be action guarded, i.e., immediately preceded by an action.

```
process Cashier() {
    do{:: get_prod palt {
                    :49: cash
                    : 1: set_price;
                         cash }
        }
}
```

C. Exception handling

Another uncommon but very serviceable language construct is the possibility to raise and handle exceptions. To illustrate this concept, we slightly adapt the description of the cashier as depicted below. In case a product cannot be cashed due to an absent price tag, the cashier calls for assistance by raising an exception (modelled by action *no_price* of exception type). On handling this exception the price is determined and the product is cashed.

```
process Cashier() {
    do{:: try { get_prod palt {
                  :49: cash
                  : 1: throw(no_price) }
              }
              catch no_price {
                  set_price;
                  cash }
        }
}
```

In a construct like try { P } catch e { Q } the body P in general models the normal behaviour, whereas if action e occurs while executing P, an exception is raised that shall be handled by Q, i.e., control is passed from P to Q. Note that compared to our previous specification, an additional action (of exception type) has been introduced to signal the occurrence of the exceptional situation.

D. Adding real-time constraints

So far, our descriptions were timeless, i.e., we did not include any timing considerations with respect to the activities involved. In the next step, we will put some simple timing constraints on the cashier. Like in timed automata [2], the elapse of time in MoDeST is modelled by means of clock variables. Values of clock variables increase linearly as time progresses. For instance, in order to impose a delay of at least 120 time units between catching the exception *no_price* and determining the price of the product at hand (*set_price*), we equip the previous description with clock variable y, and obtain the following process:

```
process Cashier() {
    do{:: try { get_prod palt {
                      :49: cash
                      : 1: throw(no_price) }
                  }
                  catch no_price {
                      y = 0;
                      when(y ≥ 120)
                          set_price;
                      cash }
        }
}
```

Clock y is reset just after catching the exception *no_price* and the price can be determined at any time point after a delay of at least 120 time-units as indicated by the when-clause. In fact, each action needs to be preceded by a when() constraint, but unless otherwise specified when(**true**) is a default constraint (that can be omitted).

E. Hard real-time constraints

When-clauses thus indicate when a certain action may (i.e. is allowed to) happen. Similar to location invariants in safety timed automata [15] and deadlines in timed automata with deadlines [5], we need a separate mechanism to *force* certain actions to happen at some time instant. To that end, we use deadlines. For instance, the process

```
process Cashier() {
    do { try { get_prod palt {
                      :49: cash
                      : 1: throw(no_price) }
                  }
                  catch no_price {
                      y = 0;
                      urgent(y ≥ 240)
```

```
            when(y ≥ 120)
               set_price;
            cash }
      }
}
```

specifies that *set_price* is enabled from 120 time units after catching the exception (as before), and that it should happen before 240 time units after the catch – as indicated by the urgent-clause. More precisely, if the exception is caught at time t, say, then *set_price* will happen at some time instant $t+\Delta$ where Δ is nondeterministically chosen from the closed interval [120,240]. Thus, differences in guards and deadline constraints induce delay nondeterminism.

In general, if an action is guarded by urgent(B), for boolean expression B, it must be executed as soon as B becomes true. Therefore, a system is allowed to idle as long as none of its activities becomes urgent. The language user can influence whether by convention activities are assumed to be urgent (guarded by urgent(**true**)), or non-urgent (guarded by urgent(**false**)), via setting a flag in the preamble of a MoDeST specification.

F. Soft real-time constraints

As a next step, we impose a delay on the cashing of the cashier, i.e., on action *cash*. Depending on (the price of) the product, environmental circumstances (such as the mood of the cashier, the time of the day), and so on, the duration of cashing may vary. We assume that cashing takes between 10 and 20 time-units. If no more information is available this could be modelled in a similar way as we just treated *set_price*. However, we now assume that the duration of cashing is uniformly distributed over the interval $[10, 20]$. In this case, the modelling as just above does not suffice, as it would choose a time instant nondeterministically without taking the likelihoods into account. To that end, we equip the specification with a clock variable x, say, and add a float variable xd, say, that is used to store a sample value drawn from a probability distribution. Thus, the occurrences of *cash* in process *Cashier* is replaced by invoking a process *Cashing* depicted below. In the latter, the statement [...] contains a set of assignments that are executed atomically, i.e., without interference with executions of other processes in the system. In this example, the variable xd is assigned a (float) value according to a uniform distribution on interval $[10, 20]$, and clock x is reset. The urgent- and when-clause make sure that *cash* takes place as soon as x has reached the value xd.

```
process Cashier() {
   do {:: try { get_prod palt {
                  :49: Cashing()
                  : 1: urgent(true)
                          throw(no_price) }
            }
            catch no_price {
                  y = 0 ;
                  urgent(y ≥ 240)
                  when(y ≥ 120)
                     set_price;
                  Cashing() }
      }
}

process Cashing() {
   [xd = U[10, 20] , x = 0];
   urgent(x ≥ xd)
   when(x ≥ xd)
      cash
}
```

G. Overall system specification

The overall system could be modelled by, for instance:

```
exception no_price;
clock x, y;
float xd;
patient get_prod, cash, set_price;

par {
   :: Arrivals() ;
   :: Queue(N) ;
   :: Cashier()
}
```

where N is the parameter (i.e., the length) of the queue. Variables do not need to be declared globally, a variable (or action, or exception) can equally well be declared local to a process. Processes are put in parallel via the par{::...} construct. These processes execute their activities independently from each other, except that common (non-local) actions need to be executed synchronously, à la CSP [16]. One of the keywords appearing in the preamble needs further explanation. We distinguish patient and impatient actions. If a patient action is common to multiple processes, then the synchronized action becomes urgent as soon as *all* partners require urgency. In contrast, a process

that intends to synchronise on an impatient action is not willing to wait for the partner. Thus a synchronized impatient action is urgent as soon as *at least one* synchronization partner requires urgency.

IV. Syntax and semantics

A. Syntax

The set of processes of MoDeST is given by the following grammar:

$$
\begin{array}{rll}
P ::= & \mathsf{stop} & \mid \ \mathit{ProcName}(e_1,\ldots,e_k) \\
\mid & \mathsf{error} & \mid \ \mathsf{alt}\{::P_1 \ \ldots \ ::P_k\} \\
\mid & \mathsf{when}(b)\ P & \mid \ \mathsf{urgent}(b)\ P \\
\mid & a\ \mathsf{palt}\ \{:w_1:asgn_1;\ P_1\ \ldots\ :w_k:asgn_k;\ P_k\} & \\
\mid & a & \mid \ \mathsf{throw}(excp) \\
\mid & \mathsf{try}\{P\}\ \mathsf{catch}\ ex_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ ex_k\ \{P_k\} & \\
\mid & \mathsf{break} & \mid \ \mathsf{do}\{::P_1 \ \ldots \ ::P_k\} \\
\mid & P_1;\ P_2 & \mid \ \mathsf{par}\{::P_1 \ \ldots \ ::P_k\} \\
\mid & \mathsf{hide}\{a_1,\ldots,a_k\}\ P & \\
\mid & \mathsf{extend}\{a_1,\ldots,a_k\}\ P & \\
\mid & \mathsf{relabel}\ \{a_1,\ldots,a_k\}\ \mathsf{by}\ \{a'_1,\ldots,a'_k\}\ P &
\end{array}
$$

where, w_i is a positive integer representing a weight, a, a'_i are (either patient, impatient or invisible) actions, a_i is an (patient or impatient) action, ex, ex_i are exceptions, b is a boolean expression, e_i is an expression not containing random variables, and $asgn_i$ is a list of assignments of the form $[x_1 = e_1,\ x_2 = e_2,\ \ldots\ ,\ x_n = e_n]$ where x_i is a variable. A MoDeST process is defined by

$$\mathsf{process}\ \mathit{ProcName}(t_1\ x_1,\ldots,t_k\ x_k)\ \{dcl\ P\}$$

where t_i is a valid type, dcl is a sequence of declarations possibly including process definitions, *ProcName* is a process name and P is as before. Type indications are often omitted for convenience.

B. Some handy shorthands

MoDeST provides some further useful operations which are shorthand notations for some common constructions. For instance, both alt and do allow an else alternative (as in Promela). else is a shorthand that can be calculated at compile time, e.g.,

$$
\begin{array}{rl}
& \mathsf{alt}\{::\mathsf{when}(b)\ P\ ::\mathsf{when}(b')\ P'\ ::\mathsf{else}\ Q\} \\
= & \mathsf{alt}\{::\mathsf{when}(b)\ P\ ::\mathsf{when}(b')\ P'\ ::\mathsf{when}(\neg(b \vee b'))\ Q\}.
\end{array}
$$

In a probabilistic alternative, either assignments or processes (but not both) can be omitted, e.g., a palt $\{:1\!:\ [y = 3]\ :2\!:\ PN(4)\ \}$ should be interpreted as a palt $\{:1\!:\ [y = 3]\ \surd\ :2\!:\ [\ ]\ PN(4)\ \}$. Notice however that, strictly speaking, the last process is not a legal MoDeST expression since $\surd$ is not in the language. The following shorthands for assignment are also allowed in MoDeST:

$$
\begin{array}{rl}
& [x = e,\ y = e'] \\
= & \mathsf{urgent}(\mathbf{true})\ \mathsf{tau}\ \mathsf{palt}\ \{:1:[x = e,\ y = e']\surd\}
\end{array}
$$

and

$$x = e = [x = e].$$

Furthermore, invariants like in safety timed automata [15] can be defined by

$$\mathsf{invariant}(b)P = \mathsf{urgent}(\neg b)\mathsf{when}(b)P.$$

MoDeST also provides other useful forms of relabelling apart from relabel and hide, and standard programming constructs are provided, such as:

$$\mathsf{while}(b)\{P\} = \mathsf{do}\{::\mathsf{when}(b)\ \ P\ ::\mathsf{else}\ \mathsf{break}\}.$$

This defines the behavioural part of MoDeST. The data part is described in [18]. In a nutshell, we allow simple and structured data types, and modularization (packages). Object-oriented enhancements (classes, sub-typing, polymorphism) are under development.

C. Semantics

The formal interpretation of a MoDeST specification is defined in terms of a state transition diagram which is extended with clock variables in order to keep track of the passage of time. This is similar to the popular model of timed automata for real-time systems [2]. Our model integrates timed automata [2] (using the deadline style of [5]), stochastic automata [9], [8], and simple probabilistic automata [23]. These three models have been carefully selected from a wide range of possible alternative models. They were chosen because they complement each other very well and yield precisely the desired expressiveness as discussed in Section II. Due to their individual compositional properties, the resulting model is elegant to come up with a compositional semantics for MoDeST.

In a nutshell, the model has the following ingredients. Locations represent the current state of the system (that includes the values of variables and clocks), while edges indicate how the system can evolve from

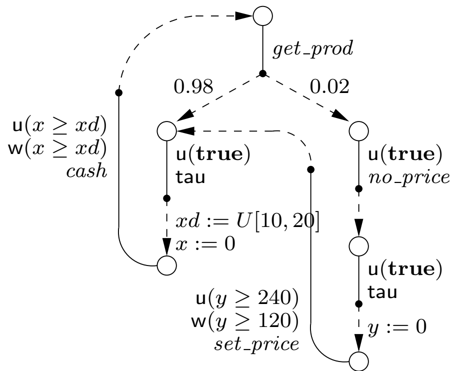

Fig. 1. A stochastic timed automaton of the cashier

one location to another. The mapping of MoDeST expressions onto these automata, called stochastic timed automata, is defined in [10]. Edges are labelled with three attributes: an *action* that is offered once the edge is executed, a *guard*, and a *deadline*. Guards and deadlines are both logical expressions over variables, possibly including clock variables. An edge labelled with guard g outgoing from location l is enabled if g holds given the current values of the variables. If in addition the deadline d holds, then the system is obliged to execute the edge before time progresses. Due to this fact, the system is allowed to wait in location l as long as no deadline in one of its outgoing edges becomes true. Once the edge is executed, the system moves to location l' with some appropriate probability while assigning values according to an indicated assignment. An example clarifies this.

Fig.1 depicts the stochastic timed automaton that corresponds to the final *Cashier* specification of Section III. Locations are represented by circles. A probabilistic edge is represented by a solid line from which dotted arrows fan out. The solid line is labelled by the guard, deadline, and action. Each dotted arrow represents a probabilistic alternative, and is labelled with a probability value and a set of assignments. Their target is the next location. Deadlines are prefixed by a 'u' (urgent) and omitted if they are **false**, and guards are prefixed by a 'w' (when) and omitted whenever they are **true**. Trivial probabilities and empty assignments are typically also omitted.

V. Tool development

We are currently implementing a tool suite to support modeling and analysis with MoDeST. The language parser is being finalised (using the ANTLR parser generator), and we are currently working on the state space generator and state space explorer. The state space generator generates a stochastic timed automaton where parallel sub-automata are not fully expanded (i.e., interleaved) so as to diminish the state-space explosion problem. It is the intention to provide the user with a flexible application programming interface (API) at this level to link MoDeST with other state-of-the-art tools. In this way, an open tool architecture should result. More concretely, we are busy with linking to Uppaal [21] for real-time model checking and to Möbius [7] for discrete-event simulation and numerical analysis. A sketch of the tool architecture is given in Fig. 2.

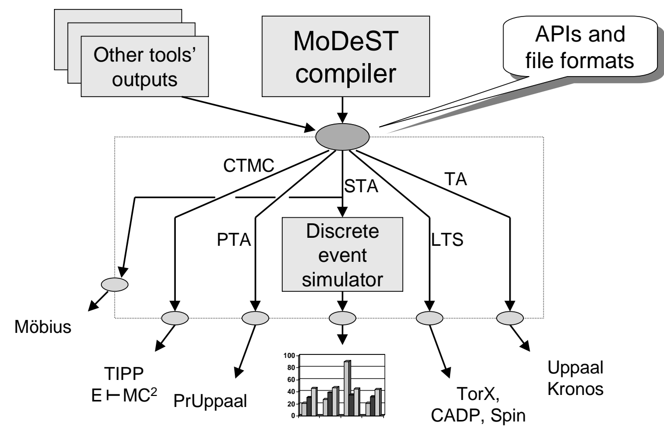

Fig. 2. Sketch of MoDeST tool architecture

VI. Conclusion

In this paper we have presented a modelling and description language that allows to combine a wide range of quantitative system aspects with functional behaviour into a single integrated specification. We gave an overview of the main rationales behind the language, and introduced the language by means of a simple, incremental example. In the next phase of the HaaST project, MoDeST and its supporting tools will be applied to the modelling and analysis of case studies of representative complexity. More information on the project can be found at `http://fmt.cs.utwente.nl/HaasT`.

Acknowledgement The authors are grateful to Ed Brinksma for inspiring discussions.

References

[1] L. de Alfaro, T.A. Henzinger and R. Majudmar. Stochastic modules. Unpublished manuscript, 1999.
[2] R. Alur and D. Dill. A theory of timed automata. *Th. Comp. Sc.*, **126**:183–235, 1994.
[3] G. Berry. Preemption and concurrency. In: R.K. Shyama-

sundar, ed, *Found. of Software Techn. and Th. Comp. Sc.*, LNCS 761, pp. 72–93. Springer-Verlag, 1993.

[4] L. Blair, T. Jones, and G. Blair. Stochastically enhanced timed automata. In: S.F. Smith and C.L. Talcott, eds, *Proc. 4th IFIP Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'00)*, pp. 327–347. Kluwer, 2000.

[5] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. and Comp.*, **163**:172–202, 2001.

[6] M. Bravetti and Gorrieri. The theory of interactive generalized semi-Markov processes. *Th. Comp. Sc.*, 2001 (to appear).

[7] D. Daly, D.D. Deavours, J.M. Doyle, P.G. Webster, and W.H. Sanders. Möbius: An extensible tool for performance and dependability modeling. In B.R. Haverkort, H.C. Bohnenkamp, and C.U. Smith, eds, *Computer Performance Evaluation*, LNCS 1786, pp. 332–336. Springer-Verlag, 2000.

[8] P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, Faculty of Computer Science, University of Twente, 1999.

[9] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In: D. Gries and W.-P. de Roever, eds, *Proc. IFIP Working Conf. on Programming Concepts and Methods*, pp. 126–147. Chapman & Hall, 1998.

[10] P.R. D'Argenio, H. Hermanns, J.-P. Katoen and J. Klaren. MoDeST – a modelling and description language for stochastic timed systems. In: L. de Alfaro and S. Gilmore, eds, *Process algebra and probabilistic methods*, LNCS 2165, Springer-Verlag, 2001.

[11] D. Ferrari. Considerations on the insularity of performance evaluation. *IEEE Trans. on Soft. Eng.*, **12**(6): 678–683, 1986.

[12] H. Garavel and M. Sighireanu. On the introduction of exceptions in E-LOTOS. In: R. Gotzhein and J. Bredereke, eds, *Formal Description Techniques IX*, pp. 469–484. Kluwer, 1996.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[14] C. Harvey. Performance engineering as an integral part of system design. *Br. Telecom Technol. J.*, **4**(3): 142–147, 1986.

[15] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. and Comp.*, **111**:193–244, 1994.

[16] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[17] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[18] R. Klaren, P.R. D'Argenio, J.-P. Katoen, and H. Hermanns. MoDeST language manual. CTIT Tech. Rep. University of Twente, 2001. To appear.

[19] J. Kramer and J. McGee. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

[20] M.Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In C. Palamadessi, ed, *Concurrency Theory*, LNCS, Springer-Verlag, 2000.

[21] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. J. of Software Tools for Technology Transfer*, **1**(1/2):134–152, 1997.

[22] J.F. Meyer, A. Movaghar, and W.H. Sanders. Stochastic activity networks: Structure, behavior and application. In: *Proc. Int. Workshop on Timed Petri Nets*, pp. 106–115, IEEE CS Press, 1985.

[23] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Dept. of Electrical Eng. and Computer Science, MIT, 1995.