MoDeST – A Modelling and Description Language for Stochastic Timed Systems

Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren

Formal Methods and Tools Group, Faculty of Computer Science
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. This paper presents a modelling language, called MoDeST, for describing the behaviour of discrete event systems. The language combines conventional programming constructs – such as iteration, alternatives, atomic statements, and exception handling – with means to describe complex systems in a compositional manner. In addition, MoDeST incorporates means to describe important phenomena such as non-determinism, probabilistic branching, and hard real-time as well as soft real-time (i.e., stochastic) aspects. The language is influenced by popular and user-friendly specification languages such as Promela, and deals with compositionality in a light-weight process-algebra style. Thus, 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.

1 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 [17,14], 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 (such as timed and probabilistic automata [1,31]). 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. In this paper, we propose a description language that is intended

L. de Alfaro and S. Gilmore (Eds.): PAPM-PROBMIV 2001, LNCS 2165, pp. 87–104, 2001.

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.

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 [21]. Data modularisation concepts and exception handling mechanisms have been adopted from modern object-oriented programming languages such as Java [16]. Process algebraic constructs have been strongly influenced by FSP (Finite State Processes [24]) 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 underspecified, 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.

[1] Exception handling in specification languages has received scant attention. Notable exceptions are Enhanced-LOTOS [15] and Esterel [3].

(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 [29,4,10,7,26]. We achieve the full expressiveness by using a model that integrates timed automata [1](using the deadline style of [6]), stochastic automata [13,11], and (simple) probabilistic automata [31]. These three ingredient models have been 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. Due to their individual compositional properties, the resulting model is elegant to use in the context of a compositonal semantics for the language MoDeST.

We claim that the language eases the description of a wide range of systems, because, in summary, it combines a rigid formal semantics with 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
- nondeterministic and probabilistic alternatives
- nondeterministic and probabilistic timing

This paper presents the formal syntax and semantics of MoDeST and discusses the relationship to existing models for probabilistic systems. The reader interested in data and data type treatments in MoDeST is referred to [22].

Organisation of the paper. Section 2 introduces the language ingredients of MoDeST in an incremental way. Section 3 defines the syntax and semantics formally. Section 4 discusses the range of models covered by MoDeST. Section 5 briefly addresses some analysis techniques for MoDeST specifications. Finally, Section 6 concludes the paper. For the sake of clarity, this paper focuses on behavioural aspects of the semantics and omits considerations on data manipulation. A full version of this paper is available [23].

2 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.

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).

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

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 depicted on the right, 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 1, **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 }
       }
}
```

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 on the right. 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.

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 [1], 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 process on the right. 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).

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

When-clauses thus indicate when a certain action may (i.e. is allowed to) happen. Similar to location invariants in safety timed automata [18] and deadlines in timed automata with deadlines [6], 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 on the right 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 catched 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.

```
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 }
    }
}
```

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.

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 on the right. In the latter, the statement $[\ldots]$ 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
}
```

The overall system could be modelled by, for instance, the expression on the right, 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 [20]. 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

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

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

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.

3 Formal Definition of MoDeST

This section formally defines the language MoDeST, the underlying operational model, and the operational semantics of MoDeST. The semantics maps each MoDeST specification on some *stochastic timed automata* (STA, for short). STA combine the power of timed automata [1] using the deadline style of [6], stochastic automata [13,11], and (simple) probabilistic automata [31]. Before discussing syntax and semantics of MoDeST we introduce STA together with other relevant concepts.

3.1 Stochastic Timed Automata

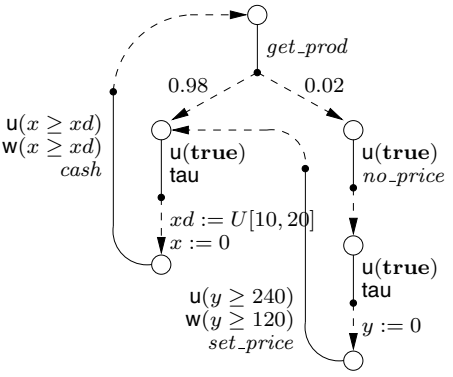
A *probability space* is a tuple $(\Omega, \mathcal{F}, \mathbf{P})$ where Ω is the *sample space*, $\mathcal{F}$ is a σ-*algebra* containing subsets of Ω, and $\mathbf{P}$ is a *probability measure* on the measurable space $(\Omega, \mathcal{F})$. If $\mathcal{P}$ is a probability space, we write $\Omega_{\mathcal{P}}$, $\mathcal{F}_{\mathcal{P}}$ and $\mathbf{P}_{\mathcal{P}}$ for its sample space, σ-algebra, and probability measure, respectively[2]. Let $Prob(H)$ denote the set of probability spaces $(\Omega, \mathcal{F}, \mathbf{P})$ such that $\Omega \subseteq H$.

Let Var be a set of typed *variables* with a distinguished subset $\mathsf{Ck} \subseteq \mathsf{Var}$ of *clock variables* (variables of type clock). Let RVar be a (finite) set of *random variables* such that $\mathsf{RVar} \cap \mathsf{Var} = \emptyset$. Let Exp be a set of *expressions* with variables in $\mathsf{Var} \cup \mathsf{RVar}$. Let $\mathsf{BExp} \subseteq \mathsf{Exp}$ be the set of *boolean expressions*, ranged over by d, d', g, $g' \ldots$. A boolean expression is required not to contain random variables. $A : \mathsf{Var} \to \mathsf{Exp}$, is called an *assignment*. Let Assign denote the set of assignments. Let Act be a set of *action names*. We use a to range over elements of Act.

Definition 1. *A* stochastic timed automaton *(STA) is a triple* $(\mathcal{S}, \mathsf{Act}, \rightarrow)$, *where* $\mathcal{S}$ *is a set of* locations *and* $\rightarrow \subseteq \mathcal{S} \times \mathsf{Act} \times \mathsf{BExp} \times \mathsf{BExp} \times Prob(\mathsf{Assign} \times \mathcal{S})$.

For $\langle s, a, g, d, \mathcal{P} \rangle \in \rightarrow$, we write $s \xrightarrow{a,g,d} \mathcal{P}$ and require that $\mathcal{P}$ is a *discrete* probability space. We call g the *guard* and d the *deadline*. Intuitively, the system is allowed to execute an edge $s \xrightarrow{a,g,d} \mathcal{P}$ whenever it is at location s and the guard g holds under 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 s as long as no deadline in one of its outgoing edges becomes true. Once the edge $s \xrightarrow{a,g,d} \mathcal{P}$ is executed, the system moves to location s' assigning values according to A with probability $\mathbf{P}_{\mathcal{P}}(\langle s', A \rangle)$.

[2] We assume familiarity with the basics of probability and measure theory (see e.g. [32]).



Depicted on the right is an example STA corresponding to the final *Cashier* specification of Section 2. Locations are represented by circles. A probabilistic edge is represented by a solid line from which doted arrows fan out. The solid line is labelled by the guard, deadline, and synchronisation label. Each dotted arrow represents a probabilistic alternative, and are 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 by a 'w' (when) and omitted whenever they are **true**. Trivial probabilities and empty assignments are also omitted.

STA provide a symbolic framework to represent stochastic timed behaviour, but this representation is too abstract to represent the concrete evolution as describe above, which is needed for different kinds of analysis, such as probabilistic model checking, or discrete event simulation. Therefore, STA have an interpretation in terms of timed continuous probabilistic transition systems. This interpretation is given in [23].

3.2 Syntax

In the following we discuss the language constructs of MoDeST. We assume that the set of actions Act consists disjointly of:

- a set PAct of *patient actions*,
- a set IAct of *impatient actions*,
- a set Excep of *exception names*,
- an action $\perp$ indicating an *unhandled error*,
- an action break indicating the *breaking of a loop*, and
- an action tau indicating an unobservable activity called *silent step*.

The set of processes of the language MoDeST is given by the following grammar,

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

where, for $1 \leq i \leq k$, w_i is a positive integer representing a *weight*, $act, act'_i \in \mathsf{PAct} \cup \mathsf{IAct} \cup \{\mathsf{tau}\}$, $act_i \in \mathsf{PAct} \cup \mathsf{IAct}$, $excp, excp_i \in \mathsf{Excep}$, $b \in \mathsf{BExp}$, $e_i \in \mathsf{Exp}$ not containing random variables, and $asgn_i$ is a list of assignments of the form $[x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n]$. A MoDeST process is defined by

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

where $\{x_1, \ldots, x_k\} \in \mathsf{Var}$, $\{t_1, \ldots, t_k\}$ are valid types, *dcl* is a sequence of declarations possibly including process definitions, *ProcName* is a process name and P is as before. We write $\mathsf{process}\ ProcName(x_1, \ldots, x_k)\ \{P\}$ instead in the remainder of this paper, for convenience.

Each set $[x_1 = e_1, \ldots, x_n = e_n]$ induces a unique assignment $A \in \mathsf{Assign}$ defined by $A(x_i) = e_i$, for $1 \leq i \leq n$, and $A(y) = y$ if $y \notin \{x_1, \ldots, x_n\}$. Therefore, we use $[x_1 = e_1, \ldots, x_n = e_n] \in \mathsf{Assign}$ to refer to its induced assignment $A \in \mathsf{Assign}$.

MoDeST provides some further useful operations which are shorthand notations for some common constructions. They are described in Appendix A.

3.3 Semantics

The operational semantics of MoDeST is defined in terms of the stochastic timed automaton $(\mathcal{S}, \mathsf{Act}, \rightarrow)$ where the set of locations $\mathcal{S}$ is defined by the set of MoDeST processes extended with a special termination mark $\surd$. The relation $\rightarrow$ is defined in the remainder of this section. In the following we use $\mathbf{Trv}(r)$ to denote the *trivial probability space* with sample space $\{r\}$. We also resort to *measurable functions*. Recall that $\mathbf{M} : \Omega_1 \to \Omega_2$ is measurable if $\mathbf{M}^{-1}(C) \in \mathcal{F}_1$ for all $C \in \mathcal{F}_2$ and that it induces a probability space $\mathbf{M}(\Omega_1, \mathcal{F}_1, \mathbf{P}_1) = (\Omega_2, \mathcal{F}_2, \mathbf{P}_1 \circ \mathbf{M}^{-1})$. In our case, all measurable functions are defined to be surjective. Under this condition $\Omega_2 = \mathbf{M}(\Omega_1)$.

Primitive operators. stop does not perform any activity and as such it does not produce any transition. *act* performs action *act* with no restriction and then terminates. break, used to break a do loop, can perform action break with no restriction and then terminates. error is a process that indicates an unhandled error by persistent executions of action $\bot$. The last of the basic operations, throw(*excp*), raises an exception by executing action $excp \in \mathsf{Excep}$. If it is not handled, the system ends up in an unhandled error. In all these cases, urgency of the execution depends on a global boolean variable urge which can be set to **true** or **false** in the preamble section of the specification. If set to **true**, the specified system responds to *maximal progress* (default is **false**). We get:

$$act \xrightarrow{act,\mathbf{true},\mathsf{urge}} \mathbf{Trv}(\surd) \qquad \mathsf{error} \xrightarrow{\bot,\mathbf{true},\mathsf{urge}} \mathbf{Trv}(\mathsf{error})$$

$$\mathsf{break} \xrightarrow{\mathsf{break},\mathbf{true},\mathsf{urge}} \mathbf{Trv}(\surd) \qquad \mathsf{throw}(excp) \xrightarrow{excp,\mathbf{true},\mathsf{urge}} \mathbf{Trv}(\mathsf{error})$$

Probabilistic prefix. act $\mathsf{palt}\ \{:w_1{:}asgn_1;\ P_1\ \ldots\ :w_k{:}asgn_k;\ P_k\}$ performs action act with no restriction, but as urgently as indicated by urge. Simultaneously, it randomly selects an alternative $i \in \{1,\ldots,k\}$ according to the weights $w_1,\ldots,w_k$, performs an assignment according to $asgn_i$, and continues executing P_i.

$$act\ \mathsf{palt}\ \{:w_1{:}asgn_1;\ P_1\ \ldots\ :w_k{:}asgn_k;\ P_k\} \xrightarrow{act,\mathbf{true},\mathsf{urge}} \mathcal{P}$$

where $\mathcal{P}$ is a discrete probability space with $\Omega_{\mathcal{P}} = \{\langle asgn_i, P_i\rangle \mid 1 \le i \le k\}$ and

$$\mathbf{P}_{\mathcal{P}}(\langle asgn_i, P_i\rangle) \stackrel{\text{def}}{=} \frac{w_i \cdot \#\{j \mid 1 \le j \le k \wedge asgn_i = asgn_j, P_i = P_j\}}{\sum_{j=1}^{k} w_j}$$

Conditions. $\mathsf{when}(b)\ P$ restricts the next activity of P to be performed whenever b holds. $\mathsf{urgent}(b)\ P$ enforces P to be urgent whenever b holds:

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

Choice. $\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}$ executes precisely one P_i, selected in a nondeterministic fashion:

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

Loop. $\mathsf{do}\{::P_1\ \ldots\ ::P_k\}$ repeatedly chooses a nondeterministic alternative. The execution finishes when one of the processes executes a break. We define the semantics of do in terms of alt and an auxiliary operator auxdo:

$$\mathsf{do}\{::P_1\ \ldots\ ::P_k\} \stackrel{\text{def}}{=} \mathsf{auxdo}\{\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}\}\{\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}\}$$

The semantics of auxdo is given by:

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad (a \neq \mathsf{break})}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathbf{M}_{\mathsf{do}}(\mathcal{P})} \qquad \frac{P \xrightarrow{\mathsf{break},g,d} \mathcal{P}}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{\mathsf{tau},g,d} \mathcal{P}}$$

where $\mathbf{M}_{\mathsf{do}}(\langle A, P'\rangle) \stackrel{\text{def}}{=} \langle A, \mathsf{auxdo}\{P'\}\{Q\}\rangle$, if $P' \neq \surd$, and otherwise, $\mathbf{M}_{\mathsf{do}}(\langle A, \surd\rangle) \stackrel{\text{def}}{=} \langle A, \mathsf{auxdo}\{Q\}\{Q\}\rangle$.

Exception handling. The process $\mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\}$ executes P and terminates if P terminates without raising any exception beforehand. If instead P raises an exception $excp_i$, it is handled by executing the respective process P_i:

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad (a \notin \{excp_1,\ldots,excp_k\})}{\mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\} \xrightarrow{a,g,d} \mathbf{M}_{\mathsf{try}}(\mathcal{P})}$$

$$\frac{P \xrightarrow{excp_i,g,d} \mathcal{P} \qquad (1 \le i \le k)}{\mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\} \xrightarrow{\mathsf{tau},g,d} \mathbf{Trv}(P_i)}$$

Table 1. Alphabet of a MoDeST term

$\alpha(\mathsf{stop}) = \alpha(\mathsf{error}) = \alpha(\mathsf{break}) = \alpha(\mathsf{throw}(excp)) = \emptyset$
$\alpha(act) = \{act\} - \{\mathsf{tau}\}$
$\alpha(act\ \mathsf{palt}\ \{:w_1{:}asgn_1;\ P_1\ \ldots\ :w_k{:}asgn_k;\ P_k\}) = \alpha(act) \cup \bigcup_{i=1}^{k} \alpha(P_i)$
$\alpha(\mathsf{when}(b)\ P) = \alpha(\mathsf{urgent}(b)\ P) = \alpha(P)$
$\alpha(\mathsf{alt}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{do}\{::P_1\ \ldots\ ::P_k\}) = \alpha(\mathsf{par}\{::P_1\ \ldots\ ::P_k\}) = \bigcup_{i=1}^{k} \alpha(P_i)$
$\alpha(P_1;\ P_2) = \alpha(P_1) \cup \alpha(P_2)$
$\alpha(\mathsf{try}\{P\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\}) = \alpha(P) \cup \bigcup_{i=1}^{k} \alpha(P_i)$
$\alpha(\mathsf{hide}\{act_1, \ldots, act_k\}\ P) = \alpha(P) - \{act_1, \ldots, act_k\}$
$\alpha(\mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{act'_1, \ldots, act'_k\}\ P) = \alpha(P)[act_1/act'_1, \ldots, act_k/act'_k] - \{\mathsf{tau}\}$
$\alpha(\mathsf{extend}\{act_1, \ldots, act_k\}\ P) = \alpha(P) \cup \{act_1, \ldots, act_k\}$
$\alpha(ProcName(e_1, \ldots, e_k)) = \alpha(P)$ provided $\mathsf{process}\ ProcName(x_1, \ldots, x_k)\ \{P\}$

where $\mathbf{M}_{\mathsf{try}}(\langle A, P'\rangle) \stackrel{\text{def}}{=} \langle A, \mathsf{try}\{P'\}\ \mathsf{catch}\ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ excp_k\ \{P_k\}\rangle$, if $P' \neq \surd$, and $\mathbf{M}_{\mathsf{try}}(\langle A, \surd\rangle) \stackrel{\text{def}}{=} \langle A, \surd\rangle$.

Sequential composition. $P_1;\ P_2$ executes P_1 until it finishes. Then it continues with the execution of P_2:

$$\frac{P_1 \xrightarrow{a,g,d} \mathcal{P}}{P_1;\ P_2 \xrightarrow{a,g,d} \mathbf{M}_;(\mathcal{P})}$$

where $\mathbf{M}_;(\langle A, P'\rangle) \stackrel{\text{def}}{=} \langle A, P';\ P_2\rangle$, if $P' \neq \surd$, and $\mathbf{M}_;(\langle A, \surd\rangle) \stackrel{\text{def}}{=} \langle A, P_2\rangle$.

Parallel composition. $\mathsf{par}\{::P_1\ \ldots\ ::P_k\}$ executes processes $P_1, \ldots, P_k$ concurrently, synchronising them on the intersected alphabet, therefore allowing multi-way synchronisation. The alphabet of a process P is the set $\alpha(P) \subseteq \mathsf{PAct} \cup \mathsf{IAct}$ of all actions P recognises. It is formally defined in Table 1. To define the semantics of MoDeST parallel composition, we resort to the auxiliary operator $||_B$, with $B \subseteq \mathsf{PAct} \cup \mathsf{IAct}$, that behaves like CSP or LOTOS parallel composition [20,5]. Thus, par is defined by

$$\mathsf{par}\{::P_1\ \ldots\ ::P_k\} \stackrel{\text{def}}{=} (\ldots((P_1\ ||_{B_1}\ P_2)\ ||_{B_2}\ P_3)\ldots)\ ||_{B_{k-1}}\ P_k$$

with $B_j = (\bigcup_{i=1}^{j} \alpha(P_i)) \cap \alpha(P_{j+1})$. The behaviour of $||_B$ is formally defined by the following rules (we omit the symmetric rule of interleaving):

$$\frac{P_1 \xrightarrow{a,g,d} \mathcal{P} \quad (a \notin B)}{P_1\ ||_B\ P_2 \xrightarrow{a,g,d} \mathbf{M}_{\mathsf{par}P_2}(\mathcal{P})} \qquad \frac{P_1 \xrightarrow{a,g_1,d_1} \mathcal{P}_1 \quad P_2 \xrightarrow{a,g_2,d_2} \mathcal{P}_2 \quad (a \in B)}{P_1\ ||_B\ P_2 \xrightarrow{a,g_1 \wedge g_2, d_1 \diamond d_2} \mathbf{M}_{\mathsf{par}}(\mathcal{P}_1 \times \mathcal{P}_2)}$$

where $d_1 \diamond d_2 = d_1 \wedge d_2$ if $a \in \mathsf{PAct}$ (that is, if the synchronising action is patient) and $d_1 \diamond d_2 = d_1 \vee d_2$ otherwise (impatient). The operator $\times$ denotes the usual product on probabilistic spaces, and $\mathbf{M}_{\mathsf{par}P_2}(\langle A, P' \rangle) \stackrel{\text{def}}{=} \langle A, P' \,||_B\, P_2 \rangle$, if $P' \neq \surd$ or $P_2 \neq \surd$, otherwise $\mathbf{M}_{\mathsf{par}\surd}(\langle A, \surd \rangle) \stackrel{\text{def}}{=} \langle A, \surd \rangle$. Furthermore,

$$\mathbf{M}_{\mathsf{par}}(\langle A_1, P_1' \rangle, \langle A_2, P_2' \rangle) \stackrel{\text{def}}{=} \begin{cases} \textbf{if } A_1 \cup A_2 \text{ is not a function } \textbf{then} & \\ \quad \langle \emptyset, \mathsf{throw}\ \mathit{inconsistency} \rangle & \\ \textbf{else} & \\ \quad \langle A_1 \cup A_2, P_1' \,||_B\, P_2' \rangle & \textbf{if } P_1' \neq \surd \text{ or } P_2' \neq \surd \\ \quad \langle A_1 \cup A_2, \surd \rangle & \textbf{if } P_1' = P_2' = \surd \end{cases}$$

Some remarks are in order. A parallel composition terminates whenever all its components terminate. Moreover, notice that the difference between synchronisation of patient and impatience actions is only given by the way the deadlines are related. Since a process that wants to synchronise on a patient action always waits for its partner to be ready, then its deadline needs to be relaxed to the requirements of the partner. As a consequence, a deadline in a patient synchronisation is met whenever all the components meet their respective deadlines. Instead, a process that intends to synchronise on an impatient action is not willing to wait for the partner. Therefore, a deadline in an impatient synchronisation should be met as soon as one of the one of the synchronising components meets its deadlines. Finally, remark that during synchronisation an inconsistency of assignments may arise due to different write accesses to the same variable, i.e., if $A_1(x) \neq A_2(x)$ for some variable x. We treat this situation by raising a predefined exception.

Relabelling and hiding. $\mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{act'_1, \ldots, act'_k\}\ P$ behaves like P except that every action act_i is renamed by the corresponding act'_i:

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad f = [act_1/act'_1, \ldots, act_k/act'_k]}{\mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{act'_1, \ldots, act'_k\}\ P \xrightarrow{f(a),g,d} \mathbf{M}_{\mathsf{relabel}}(\mathcal{P})}$$

where $\mathbf{M}_{\mathsf{relabel}}(\langle A, P' \rangle) \stackrel{\text{def}}{=} \langle A, \mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{act'_1, \ldots, act'_k\}\ P' \rangle$, if $P' \neq \surd$, otherwise $\mathbf{M}_{\mathsf{relabel}}(\langle A, \surd \rangle) \stackrel{\text{def}}{=} \langle A, \surd \rangle$.

Hiding is a particular form of relabeling in which actions are renamed by the silent action tau. Therefore we define:

$$\mathsf{hide}\{act_1, \ldots, act_k\}\ P \stackrel{\text{def}}{=} \mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{\underbrace{\mathsf{tau}, \ldots, \mathsf{tau}}_{k \text{ times}}\}\ P$$

Alphabet extension. extend is only used to extend the alphabet that a process recognises (see Table 1). Otherwise, it does not affect the behaviour:

$$\frac{P \xrightarrow{a,g,d} \mathcal{P}}{\mathsf{extend}\{act_1, \ldots, act_k\}\ P \xrightarrow{a,g,d} \mathbf{M}_{\mathsf{extend}}(\mathcal{P})}$$

where $\mathbf{M}_{\mathsf{extend}}(\langle A, P'\rangle) \stackrel{\text{def}}{=} \langle A, \mathsf{extend}\{act_1, \ldots, act_k\}\ P'\rangle$, and $\mathbf{M}_{\mathsf{extend}}(\langle A, \surd\rangle) \stackrel{\text{def}}{=} \langle A, \surd\rangle$.

Process instantiation. Provided **process** *ProcName*$(x_1, \ldots, x_k)$ $\{P\}$ is part of the MoDeST specification under consideration, *ProcName*$(e_1, \ldots, e_k)$ behaves like P where variables $x_1, \ldots, x_k$ are substituted by their respective instantiations $e_1, \ldots, e_k$.

$$\frac{P[x_1/e_1, \ldots, x_k/e_k] \xrightarrow{a,g,d} \mathcal{P}}{\mathit{ProcName}(e_1, \ldots, e_k) \xrightarrow{a,g,d} \mathcal{P}} \quad \text{provided } \mathsf{process}\ \mathit{ProcName}(x_1, \ldots, x_k)\{P\}$$

In summary, the relation $\rightarrow$ is the least relation satisfying the above rules. The reader is invited to check that the STA depicted in Section 3.1 is derived from the final *Cashier* specification of Section 2 using these semantic rules (see Appendix A for the shorthand notations used).

4 Derivable Models

MoDeST is expressive enough to cover a wide range of timed, probabilistic, nondeterministic, and stochastic models. These submodels play a crucial role in the context of analysing MoDeST specifications. Table 2 lists a range of prominent models and makes precise which semantic concepts (cf. Section 1) each of them shares with STA.

LTS: Labelled transition systems are the basic models of concurrency, they are usually analysed with techniques such as model checking or equivalence checking. They arise from MoDeST by disallowing the use of all time and stochastic concepts.

PTS: Probabilistic transition systems are labelled transition systems where some state changes are governed by discrete probability distributions while others are nondeterministic. They can be analysed with techniques from Markov decision theory, model checking, and equivalence checking [31,9]. MoDeST subsumes (simple) PTS via the **palt** construct which is action guarded by default.

Table 2. Submodels of STA

	LTS	PTS	TA	PTA	MC	GSMP	IMC	SA	STA
probabilistic branching	NO	YES	NO	YES	YES	YES	YES	YES	YES
clocks	NO	NO	YES	YES	RESTRICTED	YES	RESTRICTED	YES	YES
random variables	NO	NO	NO	NO	EXP. DIST.	YES	EXP. DIST.	YES	YES
delay nondeterminism	NO	NO	YES	YES	NO	NO	NO	NO	YES
action nondeterminism	YES	YES	YES	YES	NO	NO	YES	YES	YES

TA: Timed automata are transition systems incorporating an explicit notion of real time, represented by continuously moving clocks. Reachability analysis and model checking are the usual techniques employed for TA [1,18]. Timed automata (with deadlines) arise from MoDeST by abstaining from the use of random variables and palt.

PTA: Probabilistic timed automata are integrating TA and PTS, thus they arise from STA if random variables are unused. Reachability analysis and model checking have been proposed for PTA [25].

MC: Continuous time Markov chains are a standard model in contemporary performance evaluation. An MC is stochastic process where each delay is governed by some exponential distributed random variable. Analysis techniques for MCs range from the numerical computation of transient and steady state probabilities to approximate model checking [33,2]. MoDeST allows one to model MC by using clocks and exponentially distributed random variables, but in a restricted form (guards are right-continuous and clocks can be uniquely mapped on the random variables they use). Action and delay nondeterminism is not allowed. The model is not closed w.r.t. the operators of the language, e.g. the parallel composition of two MCs is not necessarily a MC (but an IMC).

IMC: Interactive Markov chains are MCs where action nondeterminism can occur. Therefore the model is closed w.r.t. the operators of MoDeST. An IMC can be analysed with algorithms developed for continuous time Markov decision processes [30], or sometimes be reduced to a MC by factoring the model with respect to a weak equivalence [19]. As with MCs these models can be reconstructed from a given STA, if the latter obeys certain restrictions. MoDeST provides shorthand notations making it possible to ensure these restrictions by default: A specification where stochastic aspects only make use of these shorthands possesses a direct semantics in terms of IMC (without reconstructing the latter from the STA semantics).

GSMP: Generalized semi-Markov processes are a general purpose performance evaluation model. Theses stochastic processes are usually analysed using discrete event simulation, but in specific cases a numerical analysis is also feasible. GSMPs arise from MoDeST specifications if action and delay nondeterminism does not occur. The model is not closed w.r.t. the operators of the language, e.g. the parallel composition of two GSMPs is not necessarily an GSMP (but a SA).

SA: Stochastic automata are basically GSMPs with action nondeterminism (hence they are closed under composition), but can also be seen as TA where delay nondeterminism is replaced by random variables governing the delays [13, 11]. As with IMC, specific shorthands can be used to ensure the restrictions required to obtain a SA. For instance if X is a random variable then $\mathsf{wait}(X)$ is an abbreviation for $[x = X\,; c = 0]\ \mathsf{urgent}(c \geq x)\ \mathsf{when}(c \geq x)\ \mathsf{tau}$ where c

(respectively x) is a clock variable (float variable) private to X. Again, these shorthands are used to map the MoDeST specification directly on the SA which otherwise is retrievable from the STA semantics.

It is important to remark that the presence of each listed semantic concept – apart from action nondeterminism – can be detected syntactically, while parsing a specification. This is trivial for probabilistic branching (palt), and obvious for clocks, because they have to be declared before use in MoDeST. Use of random variables is easily detected while parsing because (exponential or general) continuous probability distributions are provided via a predefined class (i.e., type). Delay nondeterminism is absent in a specification if for each action the guard and deadline agree. So, Table 2 also gives sufficient syntactic criteria for identifying submodels while parsing a MoDeST specification.

Action nondeterminism is a principal feature for compositional formalisms, yet it induces that MCs and GSMPs are not closed under composition in general. Action nondeterminism can in principle be excluded syntactically by disallowing alt and par, but the resulting language is too meager to be of much use. More liberal syntactic conditions for absence of action nondeterminism can be adopted from [28].

5 Model Analysis

The identification of well-studied submodels is of crucial practical relevance, because the enormous expressiveness of MoDeST comes with the drawback that the underlying general model is not well investigated: So far analysis methods for the general STA model have not been devised, and their development is ongoing work. The general idea behind this work is strongly based on the identification of submodels of STA for which analysis methods have been published. Based on this knowledge, four different strands can be pursued:

- Isolate syntactic subclasses of MoDeST that map on well-investigated submodels. As long as the user of MoDeST adheres to such a subset, the proper analysis engine can be determined mechanically.
- Define abstractions from STA to less specific models. One such abstraction [12] is to mask the distributions of random clocks i.e., to consider random clocks as delay nondeterministic clocks. In this way, any STA can be turned into a TA by abstracting the stochastic behaviour. Real-time model checking on this TA is safe w.r.t. to the original STA model.
- Define concretisations from more general models to more specific models. This usually means to add additional explicit modelling assumptions, such as to assume a particular scheduler to resolve action nondeterminism, or to assume that all random clocks follow an exponential or phase-type distribution. Note that the quantitative error introduced by such an assumption can be unbounded in certain circumstances.
- Extend or combine analysis methods from submodels of STA to full STA. In particular we are planning to integrate real-time model checking of TA with numerical recipes for GSMPs.

6 Conclusion

In this paper we have introduced a modelling and description language for stochastic timed systems. We have formally defined syntax and semantics of MoDeST, and have put the language in the context of other well-studied models. The focus of this paper has been the behavioural part of MoDeST. The data part is described in [22]. In a nutshell, we allow simple and structured data types, and modularization (packages). Object-oriented enhancements (classes, sub-typing, polymorphism) are under development.

We are currently implementing a tool suite to support modeling and analysis with MoDeST. The language parser is being finalised, and we are working on the state space generator now. The main strategy we pursue in this respect is to bridge to state-of-the-art verification and analysis tools on the level of the STA model. More concretely, we are busy with linking to UPPAAL [27] for real-time model checking and to MÖBIUS [8] for discrete event simulation and numerical analysis.

Acknowledgement The authors are grateful to Ed Brinksma for inspiring discussions. This work is supported by grant TES-4999 of the Dutch Technology Foundation (STW) and grant 612.069.001 of the Netherlands Organisation of Scientific Research (NWO).

A Further MoDeST Expressions

MoDeST provides operations which are shorthands 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.,

$$\begin{aligned} & \mathsf{alt}\{::\mathsf{when}(b_1)\ P_1\ \ldots\ ::\mathsf{when}(b_k)\ P_k\ ::\mathsf{else}\ Q\} \\ \stackrel{\text{def}}{=}\ & \mathsf{alt}\{::\mathsf{when}(b_1)\ P_1\ \ldots\ ::\mathsf{when}(b_k)\ P_k\ ::\mathsf{when}(\neg \textstyle\bigvee_{i=1}^{k} b_i)\ Q\}. \end{aligned}$$

In a probabilistic alternative, either assignments or processes (but not both) can be omitted, e.g., *act* palt {:1: $[y = 3]$:2: $PN(4)$ } should be interpreted as *act* 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:

$$\begin{aligned} [x_1 = e_1, \ldots, x_k = e_k] &\stackrel{\text{def}}{=} \mathsf{urgent}(\mathbf{true})\ \mathsf{tau}\ \mathsf{palt}\ \{: 1 : [x_1 = e_1, \ldots, x_k = e_k]\surd\} \\ x = e &\stackrel{\text{def}}{=} [x = e]. \end{aligned}$$

Furthermore, invariants like in safety timed automata [18] can be defined by

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

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

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

References

1. R. Alur and D. Dill. A theory of timed automata. *Th. Comp. Sc.*, **126**:183–235, 1994.
2. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In: J.C.M. Baeten and S. Mauw, eds, *Concurrency Theory*, LNCS 1664, pp. 146–161. Springer-Verlag, 1999.
3. G. Berry. Preemption and concurrency. In: R.K. Shyamasundar, 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. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Netw. and ISDN Sys.*, **14**:25–59, 1987.
6. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. and Comp.*, **163**:172–202, 2001.
7. M. Bravetti and Gorrieri. The theory of interactive generalized semi-Markov processes. *Th. Comp. Sc.*, **258**, 2001 (to appear).
8. 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.
9. L. de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, 1997.
10. L. de Alfaro, T.A. Henzinger and R. Majudmar. Stochastic modules. Unpublished manuscript, 1999.
11. P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* PhD thesis, Faculty of Computer Science, University of Twente, 1999.
12. P.R. D'Argenio. A compositional translation of stochastic automata into timed automata. Technical Report CTIT 00-08, Faculty of Computer Science, University of Twente, 2000.
13. 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.
14. D. Ferrari. Considerations on the insularity of performance evaluation. *IEEE Trans. on Soft. Eng.*, **12**(6): 678–683, 1986.
15. 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.
16. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
17. C. Harvey. Performance engineering as an integral part of system design. *Br. Telecom Technol. J.*, **4**(3): 142–147, 1986.
18. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. and Comp.*, **111**:193–244, 1994.
19. H. Hermanns. *Interactive Markov Chains.* PhD thesis, University of Erlangen-Nürnberg, 1998.
20. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
21. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

22. R. Klaren, P.R. D'Argenio, J.-P. Katoen, and H. Hermanns. Modest language manual. CTIT Tech. Rep. University of Twente, 2001. To appear.
23. P.R. D'Argenio, H. Hermanns, J.-P. Katoen, and R. Klaren. MoDeST – a modelling and description language for stochastic timed systems. CTIT Tech. Rep., University of Twente, 2001.
24. J. Kramer and J. McGee. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.
25. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with probability distributions. In: J.-P. Katoen, ed, *Formal Methods for Real-Time and Probabilistic Systems*, LNCS 1601, pp. 75–95. Springer-Verlag, 1999.
26. 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.
27. 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.
28. V. Mertsiotakis. *Approximate Analysis Methods for Stochastic Process Algebras*. PhD thesis, University of Erlangen-Nürnberg, 1998.
29. 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.
30. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
31. R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Dept. of Electrical Eng. and Computer Science, MIT, 1995.
32. A.N. Shiryaev. *Probability*, volume 95 of *Graduate Texts in Mathematics*. Springer-Verlag, 1996.
33. W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
34. W. Yi. Real-time behaviour of asynchronous agents. In: J.C.M. Baeten and J.-W. Klop, eds, *CONCUR 90*, LNCS 458, pp. 502–520. Springer-Verlag, 1990.